

Cisco IOS Programmer's Guide/ Architecture Reference

Twentieth Edition
March 2009
Software Release 12.4

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)

Contact enged-update@cisco.com for an accessible format per
ACC-DOC-30.21.

CISCO HIGHLY CONFIDENTIAL

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The following information is for FCC compliance of Class A devices: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case users will be required to correct the interference at their own expense.

The following information is for FCC compliance of Class B devices: The equipment described in this manual generates and may radiate radio-frequency energy. If it is not installed in accordance with Cisco's installation instructions, it may cause interference with radio and television reception. This equipment has been tested and found to comply with the limits for a Class B digital device in accordance with the specifications in part 15 of the FCC rules. These specifications are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation.

Modifying the equipment without Cisco's written authorization may result in the equipment no longer complying with FCC requirements for Class A or Class B digital devices. In that event, your right to use the equipment may be limited by FCC regulations, and you may be required to correct any interference to radio or television communications at your own expense.

You can determine whether your equipment is causing interference by turning it off. If the interference stops, it was probably caused by the Cisco equipment or one of its peripheral devices. If the equipment causes interference to radio or television reception, try to correct the interference by using one or more of the following measures:

- Turn the television or radio antenna until the interference stops.
- Move the equipment to one side or the other of the television or radio.
- Move the equipment farther away from the television or radio.
- Plug the equipment into an outlet that is on a different circuit from the television or radio. (That is, make certain the equipment and the television or radio are on circuits controlled by different circuit breakers or fuses.)

Modifications to this product not authorized by Cisco Systems, Inc. could void the FCC approval and negate your authority to operate the product.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Cisco and the Cisco Logo are trademarks of Cisco Systems, Inc. and/or its affiliates in the U.S. and other countries. A listing of Cisco's trademarks can be found at www.cisco.com/go/trademarks. Third party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1005R)

Cisco IOS Programmer's Guide/Architecture Reference
Twentieth Edition March 2009
Release 12.4

Nineteenth Edition December 2008, Release 12.4
Eighteenth Edition December 2007, Release 12.4
Seventeenth Edition April 2007, Release 12.4
Sixteenth Edition July 2006, Release 12.4
Fifteenth Edition September 2005, Release 12.4
Fourteenth Edition February 2005, Release 12.3
Thirteenth Edition July 2004, Release 12.3

Twelfth Edition September 2003, Release 12.3
Eleventh Edition January 2003, Release 12.2
Tenth Edition July 2002, Release 12.2
Ninth Edition February 2002, Release 12.2
Eighth Edition September 2001, Release 12.2
Seventh Edition June 2001, Release 12.2
Sixth Edition May 2000, Release 12.1
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

Copyright © 1995-2010, Cisco Systems, Inc.
All rights reserved. Printed in USA.

1005R

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, Joel Obstfeld, Steve Glaus, Marcos Klein, Marie Godfrey, Hollis Johnson, Tony McClure, Katy Shotton, Don Banks, Alison Apel, John Lautmann, Peter Hanselmann, Leesa Stanion, Teresa Smith-Milo, Tom Gee, Paul Sandoval, Philip Prindeville, Jeff Learman, Rick Pratt, Beth Montoya, Pipson Sebastian Mampilli, Jyothi Ramadas, A.S Ashok

Editor: Susan Purcell

Change History

Email updates and change requests for this document to ios-doc@cisco.com.

Changes in the Twenty-first Edition (*Month 200x*)

The following changes have been made in the twenty-first edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software..

Chapter	Change
Chapter 1, "Overview"	Updated the Cisco IONization 101 Guidelines link and moved the "Migrating to ION" to the end of the chapter as section 1.7. (June 2010)
Chapter 2, "System Initialization"	Added additional 3750E and 3560E warm upgrade functional information to section 2.4 "IOS Warm Upgrade". (February 2010)
Chapter 3, "Basic IOS Kernel Services"	<p><i>Redrew Figure 3-1, Figure 3-2, Figure 3-3, Figure 3-4, Figure 3-5 and Figure 3-6.</i> (August 2011)</p> <p>Added more information about <code>process_may_suspend()</code> function in Table 3-2 "Functions for Checking and Suspending Tasks". (August 2010)</p> <p>Added more information about preemptive process in the section 3.3.6.3.1 "Pseudo-preemption Infrastructure Memory and Performance Considerations". (July 2010)</p> <p>Added the following macro to <code>create_watched_priority_semaphore()</code> and <code>create_watched_recursive_semaphore()</code> in section 3.4.3.6.1</p> <p>Updated <code>process_may_suspend_on_quantum()</code>, <code>process_suspend_if_req()</code>, and <code>process_time_exceeded()</code> in Table 3-2 "Functions for Checking and Suspending Tasks" to say that the use of these functions is strongly discouraged. (July 2009)</p>

Chapter 4, "Memory Management"	<p>Added steps to show an example of how to debug a packet element leak to section 4.7.6.3.3 "show memory debug leaks chunks". (October 2011)</p> <p>Added websites for more information on memory management at beginning of chapter. (October 2011)</p> <p>Added URL to section 4.7.2.3 "show memory dead Output". (August 2011)</p> <p>Added the <code>CHUNK_FLAGS_NOHEADER_FAST</code> flag to Table 4-8 "Chunk Pool Flags". (December 2010)</p> <p>Removed references to <code>MEMPOOL_CLASS_FAST</code>, <code>MEMPOOL_CLASS_ISTACK</code>, <code>MEMPOOL_CLASS_PCIMEM</code>, <code>MEMPOOL_CLASS_PSTACK</code>, and <code>MEMPOOL_CLASS_MULTIBUS</code>. (October 2010)</p> <p>Added information about Garbage Detector as section 4.7.6 "Garbage Detector". (July 2010).</p> <p>Added a note on the memory pointer setting by the application to section 4.3.10.1 "Example: Lock Memory". (June 2010)</p> <p>Corrected the space issue in the return commands (June 2010).</p> <p>Added <code>CHUNK_FLAGS_NONLAZY</code> flag (New in 12.4(20)T) to Table 4-8 "Chunk Pool Flags". Added note to section 4.4.3.1 "Example: Create a Memory Chunk". (July 2009)</p> <p>Updated <code>CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF</code> flag (New in 12.2SR(autobahn76)) in Table 4-8 "Chunk Pool Flags". (May 2009)</p>
Chapter 6, "Interfaces and Drivers"	<p>Added new section 6.17 "Default Box-Wide IEEE MAC Address". (April 2011)</p> <p>Switched section 6.2 "Manipulate IDBs" and 6.3 "Subblocks and Private Lists".</p> <p>Divided section 6.3 "Manipulate IDB Subblocks" into section 6.4 "Subblock Implementation Before Release 12.2S" and section 6.5 "Subblock Implementation After Release 12.2S". (November 2010)</p> <p>Added information related to HWIDB recycling to section 6.2.5 "Delete an IDB". (June 2010)</p> <p>Added information to clarify the ways a developer can prevent the user from accessing or viewing an IDB. This is reflected in section 6.12, "Protecting an IDB from the User." (November 2009)</p> <p>Complete overhaul of section 6.2 "Manipulate IDBs" to document the new way of creating IDBs. Changed <code>pkt_scheduling@cisco.com</code> to <code>interest-idb@cisco.com</code> in note in section 6.1 "Introduction". Changed <code>sys/h/subblock.h</code> to <code>subblock.h</code> in reference to where the globally assigned enumerator identifier is defined in section 6.3 "Subblocks and Private Lists". (October 2009)</p>
Chapter 8, "Interprocessor Communications (IPC) Services"	Added the section 8.13 "IPC Multicast Messages".(Februray 2011)
Chapter 9, "File System"	Replaced the NVRAM Multiple File System section with the section 9.7 "Persistent Variable Method".(July 2010)
Chapter 11, "Standard Libraries"	This chapter underwent a complete overhaul because of the new security guidelines. (December 2010) Revised chapter after Editorial Review. (June 2010)

Chapter 14, “Registries and Services”	Added new section 14.5.1.2 “Incorrect Macro Usage”. (April 2011) Added new section 14.19 “Manipulate SEQ_LIST Services”, which describes how to use the SEQ_LIST registry service. (November 2009) Corrected the filename in section 14.5 “Steps to Create and Use a Registry” to read <code>files_reg.mk</code> in the subsection that describes how developers add or remove registries in the Registry ReDefine Project. (October 2009) Updated the CASE type of service in section 14.1.2 “Types of Services” and section 14.10 “Manipulate CASE Services”. (April 2009)
Chapter 19, “Writing Cisco IOS Error Messages”	Added text to section 19.1 “Error Message Guidelines”. (April 2011)
Chapter 20, “Debugging and Error Logging”	Added new section 20.28 “The sdec Tool and Stack Corruption Troubleshooting”. (May 2011) Added new subsection 20.27.1 “ASLR and Impact on Debugging”. (September 2010) Added new section 20.27 “Debugging ASLR Enabled Cisco IOS Images”. (June 2010) Added new section 20.5 “Enable Debug commands during boot-up in IOS” (June 2010) Added new section 20.26 “Cisco Error Number (errno)”. (March 2010) Added new cross-reference to http://www.catb.org/~esr/faqs/smash-questions.html in section 20.7 “Links to Other Debugging Documentation”. (April 2009)
Chapter 26, “Porting Cisco IOS Software to a New Platform”	Added new section 26.1.4 “Adding a New CPU Type”. (March 2010) Added new section 26.1.5.2.1 “Reasons for Stack Overflow”. (April 2009)
Chapter 27, “Command-Line Parser”	Updated section 27.7.3 “Important Notes on Data Variables”. (October 2010) Added a note about the impact of white spaces within keywords. (October 2010) Added a new section 27.3.7.3 “Difference between NONE and no_alt”. (August 2010) Added a note in the section 27.13.1 “Add a Parser Mode”. (July 2010) Removed all parser dump utility commands from Table 27-7 “Useful show parser Commands” because these commands were deprecated. (July 2010) Added PRC error codes in the section 27.5.2.1 “Error Codes” and removed the section 27.3.7 “Usage of ALTERNATE, no_alt and NONE”. (June 2010) Added the newly created OPTIONAL_KEYWORD macro to Table 27-1 “Macros for Parsing Keywords”. (March 2010) Added new section 27.3.10 “Generating Error Messages in Configuration Mode”. (March 2010) Added new section 27.12.6 “Limiting Parser Command Searches in Interface Submodes”. (February 2010) Added additional information to section 27.8 “Ordering Commands”. (October 2009). Added new section 27.3.2.2 “How to Write Commands for Subinterfaces”. (June 2009)
Chapter 29, “MIB Infrastructure”	Section 29.2.2 “Ethernet Interface Manager” was added in 2007 but because there is no record of it in this Change History file, the record of it is added here. (March 2010)
Chapter 30, “Security”	Added a new section 30.1.6 “Asking and Verifying Passwords” that explains how passwords work. (September 2010)

Appendix A, "Writing Cisco IOS Code: Style Issues"	Added information about a code indentation checking tool in a new subpara "Checking Indentation," in sub-section A.4.1, "Specific Code Formatting Issues." (January 2011) Added the command length details as a new subpara "Command Length", in sub-section A.4.1 "Specific Code Formatting Issues". (July 2010) Added a note on __func__, __FUNCTION__ and __PRETTY_FUNCTION__ to the "Function Prototypes" sub-section of A.3.2 "Fifty Ways to Shoot Yourself in the Foot". (July 2010) Added details on using return statement with parentheses to section A.4.1 "Specific Code Formatting Issues". (June 2010) Added a note on exception to explicit typecasting to section A.3.2 "Fifty Ways to Shoot Yourself in the Foot". (March 2010) Added new section A.12 "Using Open Source Code in IOS Software". (August 2009) Added an example code, illustrating the proper usage of curly braces, spacing, and indents in section Appendix A.4.1 Specific Code Formatting Issues, sub-section Curly Braces. (April 2009)
Appendix F, "Doxygen Instructions"	Added a note related to doxygenated documents location for the API's in first page of the chapter. (May 2009)
Appendix G, "Glossary"	Added the definitions of CPUHOG, process watchdog, scheduler watchdog, and software watchdog. (July 2010)

Changes in the Twentieth Edition (March 2009)

The following changes have been made in the twentieth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software..

Chapter	Change
About This Manual	Replaced reference to Docx with reference to Doxygen in the Document Organization
Chapter 1, "Overview"	Replaced reference to Docx with reference to Doxygen in section 1.3.7.6 "Doxygen Instructions". (March 2009) Added new section "Searching Email Alias News Archives." (January 2009)
Chapter 3, "Basic IOS Kernel Services"	Added warning that the use of this function is strongly discouraged to process_may_suspend_on_quantum(), process_suspend_if_req(), and process_time_exceeded() in Table 3-2. (February 2009)
Chapter 4, "Memory Management"	Added CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF flag (new in 12.2SR(autobahn76)) to Table 4-8 "Chunk Pool Flags". (February 2009) Added new section 4.12.3 "LRU ID Manager". (February 2009) Added the following memory-related commands: show memory allocating-process [totals], show memory debug references, show memory ecc, show memory fast [allocating-process [totals] dead [totals] free [totals]], show memory fragment, show memory multibus, show memory pci, show memory processor, show memory scan, and show memory transient [allocating-process [totals] dead [totals] fragment [detail] free [totals] statistics [history]]. (February 2009)
Chapter 7, "Platform-Specific Support"	Added descriptions of allocation, storage, and use of HWIDB attributes to Section 7.15, "Interface ID Format" and common encoding examples for slotunit and subunit in Section 7.15.1, "Slotunit, Subunit, and Unit Values". (February 2008)

Chapter 8, “Interprocessor Communications (IPC) Services”	Completely revised chapter. (February 2009) Added new section 8.25 “IPC Get Port Notifications”. (February 2009)
Chapter 14, “Registries and Services”	Added the registry services, SEQ_ILIST and SEQ_LIST. (March 2009)
Chapter 15, “Time-of-Day Services”	In section 15.7, “Format Time Strings,” deprecated functions: <code>current_time_string()</code> , <code>unix_time_string()</code> , and <code>unix_time_string_2()</code> . Added new entries for functions: <code>current_time_string_safe()</code> , <code>unix_time_string_safe()</code> , and <code>unix_time_string_2_safe()</code> . (January 2009)
Chapter 27, “Command-Line Parser”	Updated the section 27.13, “Manipulating Parser Modes,” to include programming the correct behavioral operation of submode commands using the HTTP configuration interface. (February 2009)
Appendix F, “Doxygen Instructions”	Completely revised chapter (replaces docx information). (March 2009)

Changes in the Nineteenth Edition (December 2008)

The following changes have been made in the nineteenth edition of the *Cisco IOS Programmer’s Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software.

Chapter	Change
Chapter 4, “Memory Management”	Updated 4 links in the section 4.7 “Memory Management Commands”. (September 2008) Added show mem big entry in Table 4-9 “Memory-Related Commands.” (February 2008) Added new section 4.3.16 “IOS Memory Scaling.” (February 2008) Added a note that describes the variation in size of blocktype_ across various branches of IOS and how to identify the blocktype_ size, to subsection 4.7.2.1, “show memory Output.” Added note that the Garbage Detector does not work on external memory in Section 4.4.3.3, “External Memory” and added new Section 4.9.4, “Safe Bitlist API Functions”. (January 2008)
Chapter 6, “Interfaces and Drivers”	Added description of <code>idb_create_with_identity</code> in section 6.2.1, “Create an IDB”. Added new section 6.2.5, “Reuse an IDB” and modified the introductory paragraph in section 6.2, “Manipulate IDBs” to introduce the new section. (February 2008)
Chapter 7, “Platform-Specific Support”	Added descriptions of allocation, storage, and use of HWIDB attributes to Section 7.15, “Interface ID Format” and common encoding examples for slotunit and subunit in Section 7.15.1, “Slotunit, Subunit, and Unit Values”. (February 2008)
Chapter 11, “Standard Libraries”	The list of ANSI C library functions and IEEE-1003.1-2001 POSIX standard library functions in the Cisco IOS Programmer’s Guide in Table 11-3, “Available ANSI C, POSIX, and Related Library Functions” has been moved to the Cisco IOS Programmer’s Guide. (February 2008)
Chapter 16, “Timer Services”	Clarified the implications of use of <code>TIMER_EVENT</code> in subsection 16.4.4.2, “Active, Stopped, and Expired Managed Timers.” (January 2008)
Chapter 21, “Binary Trees”	Added a new section 21.3.1.5 “Remove an AVL Tree.” (October 2008)

Chapter 24, "High Availability (HA)"	Added new section "Private Configuration" to section 24.4.2.3, "IOS Infrastructure Components," to explain the mechanism by which the private configuration is sync'ed with the Standby RP. (February 2008) Added new section "Parsing at System Startup" to section 24.4.2.3.1 "Configuration Synchronization (Config Sync)". (January 2008)
Chapter 25, "IP Services"	Added <code>ip_forus()</code> and <code>ip_ouraddress()</code> to the list of functions in Section 25.1, "IP API Functions". (January 2008)
Chapter 26, "Porting Cisco IOS Software to a New Platform"	Corrected the values of <code>sword</code> and <code>lword</code> and indicated that the values were in hexadecimal format in section 26.1.1.1 "Unions". (January 2008)
Chapter 27, "Command-Line Parser"	Added section 27.1.1, "Using CLI Editing Features and Shortcuts" to indicate how to find out the currently used key bindings for the CLI. Added index entries for "ambiguity" and debug parser ambiguity. (September, 2008) Completed overhaul of Command-Line Parser chapter, after thorough review by a member of the Parser Development team. (March, 2008) Updated for consistent use of term "submode". Updated 27.12.2 "Exiting a Submode", and added 27.12.3 "Implicit Submode Exit to Parent Mode", to describe the commonly-encountered but often misunderstood implicit behavior where if the parser fails in a submode, it retries the command in the parent mode. Changed title of Section 27.11, "Useful Exit Functions", to 27.12 "Useful Exit Functions and Submode Attributes". (January 2008)
Chapter 30, "Security"	Added section 30.1.9, "Watching Multiple Queues: Avoiding DOS Attacks FAQ," to describe how to prevent DOS attacks when a process is watching multiple queues. (January 2008) Performed first pass at restructuring this chapter so that section 30.1 "Secure Coding" is followed by section 30.2 "IOS Security Features". (January 2008)
Appendix A, "Writing Cisco IOS Code: Style Issues"	Moved content of section A.10 "Coding for Security" to section 30.1 "Secure Coding". (January 2008)

Changes in the Eighteenth Edition (December 2007)

The following changes have been made in the eighteenth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software.

Chapter	Change
Chapter 2, "System Initialization"	Added new section "Checking for Interrupt Status". (November 2007)

Chapter 3, "Basic IOS Kernel Services"	Noted in section 3.4.4.1 "Directly Waking up a Cisco IOS Task", and 3.4.4.3 "Waiting on Events" that process_wakeup() and the DIRECT_EVENT wakeup reason code are being deprecated in Cisco IOS Releases 12.3 and greater. Updated Section 3.4.4.5 "Disabling Blocking and Using Safe Blocking with process_safe_() Functions" to reflect new behavior applied in CSCsj24186 to save and restore the current blocking state around the related process blocking call (the corresponding function without "safe" in its name). Tasks using these functions should no longer assume that blocking will be disabled upon returning from these function calls. Updated Section 3.1.2 "List of Functions", Section 3.2 "Scheduler", Table 3-2, and Section 3.2.4.4 "Moving Tasks between Queues" to mention the basic process scheduling functions available for relinquishing the processor: process_suspend(), process_suspend_on_quantum(), process_may_suspend(), and process_may_suspend_on_quantum(). The update was prompted by CSCeg73826, which added the process_suspend_on_quantum() function and updated process_may_suspend_on_quantum() to call process_may_suspend() instead of process_suspend(). (November 2007)
Chapter 4, "Memory Management"	Added additional cross-reference to http://wwwin-engd.cisco.com/common/courses/mem/ID-49.html#0_pgId-6617 in section 4.7.2.1.1, "The blocktype Header and **previous" (November 2007) Added section 4.3.15 "Memchecks Support". (November 2007) Added section 4.13, "Shared Information Utility" (<i>New in 12.2SX</i>). (July 2007)
Chapter 11, "Standard Libraries"	Added instructions to access The Open Group System Interfaces reference document from their website in section 11.4.2.2, "Accessing POSIX Standards Reference Documentation". (June 2007)
Chapter 15, "Time-of-Day Services"	Updated to reflect that the Cisco IOS timeval structure was renamed ios_timeval so as not to conflict with POSIX timeval structure in recently-ported standard time APIs, as follows: Noted the change in subsection 15.1.2, "Time Formats"; updated structure definition in subsection 15.1.2.3, "ios_timeval Structure"; updated function prototypes for time conversion functions in section 15.4, "Convert between Time Formats", Table 15-2, and for formatting time strings in section 15.7, "Format Time Strings". (June 2007)
Chapter 16, "Timer Services"	Added subsection 16.4.4.1, "Using Interrupt Routines and Managed Timers." (December 2007)
Chapter 17, "Strings and Character Output"	Removed the section 17.1.5, "Ask the question ---MORE---", and all references to automore_more(), an internal function documented by mistake that is not externally available; the automore_conditional() API function should be called with parameter value 0 to invoke this functionality instead. Renamed and modified section 17.1.9, "Conditionally Prompt to Do More Output" (was previously section 17.1.10, "Conditionally Ask for Permission to Do More Output"), to more clearly describe the automore_conditional() API function. Updated section 17.1.5, "Turn on Automatic ---MORE--- Processing" to include an overview of the behavior of automore processing after it is enabled with the automore_enable() API function. Similar updates to sections 17.1.6, "Change Automore's Header in Midstream", 17.1.7, "Disable ---MORE--- Processing", and 17.1.8, "Find Out if User has Quit Automore". (June 2007)
Chapter 19, "Writing Cisco IOS Error Messages,"	Updated the document number for the source of some of the information included in this chapter: now refers to EDCS ENG-77462 document. Also added the error message review email alias as a point of contact for development questions. (November 2007)

Chapter 27, “Command-Line Parser”	Added new sections: 27.20.1, “Config Checkpointing,” and 27.20.1.1, “NVGEN Config Checkpointing,” to describe the config checkpointing options available in <i>Whitneyx</i> branches. (December 2007) Added first paragraph in Section 27.7, “Ordering Commands” and moved the existing note at the end of the section to be the second paragraph. Added index entry for <code>parser:parse trees: NVGEN</code> . (November 2007) Added index entries for <code>OBJ(type,N)</code> variable and <code>OBJ:non-zero</code> during Show in section 27.3.1.4, “How to Avoid Collateral OBJ Damage in the Parse Trees”. (November 2007) Updated Section 27.3.3, “Parse a Number Token”, for the deprecation of octal number parsing (CSCesc84077) committed to 12.2S train, and the deprecation of the <code>NUMBER_STRICT</code> flag (introduced in CSCEi92592). (July 2007)
Chapter 28, “Writing, Testing, and Publishing MIBs”	Added information for the General Management Infrastructure (GMI) on the SNMP Infra Tool in sections 28.4, 28.5, and 28.6, added phase 3, “Review” to section 28.4, “MIB Life Cycle”, and a new section 28.6, “MIB Review”. (August 2007)
Chapter 34, “Verifying Cisco IOS Modular Images”	This chapter is no longer relevant and is now deprecated. (June 2007)
Appendix A, “Writing Cisco IOS Code: Style Issues”	Revised note to describe use of email aliases. (December 2007)
Appendix E, “Branch Integration & Sync Processes”	Removed references to the reset hyperlink procedure throughout the chapter. (October 2007)

Changes in the Seventeenth Edition (April 2007)

The following changes have been made in the seventeenth edition of the *Cisco IOS Programmer’s Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software.

Chapter	Change
Chapter 3, “Basic IOS Kernel Services”	Added Section 3.3.5, “Process Accounting and CPU Utilization”, to describe how process accounting is managed. Includes how to implement the enhancement introduced in CSCec17882, feature specification EDCS-314181, for more accurate CPU usage accounting. Describes API function <code>enable_correct_accounting()</code> , processor-specific macros <code>SAVE_INTR_START_TIME</code> and <code>UPDATE_INTR_CPU_USAGE</code> , and timer ISR updates to maintain more accurate usage totals for process-level and interrupt-level contexts. Added that the stack is filled with <code>0xffffffff</code> for every <code>process_create()</code> call in section 3.3.3.2, “Creating a Cisco IOS Task”. (November 2006) Added a note on the behavior of <code>process_wait_for_event()</code> when events are not handled and an example to subsection 3.4.4.3, “Waiting on Events”. (October 2006)

Chapter 4, “Memory Management”	<p>Changed Table 4-9 name from “Memory-related show Commands” to “Memory-Related Commands”. Added additional references for Memory Leak Detector, Embedded Resource Manager, and other useful memory-related commands, features, and debugging aids to Table 4-9 and section 4.7, “Memory Management Commands”. Fixed description of <code>Retbufs</code> column in Table 4-13, “show process memory Column Description”. (January 2007)</p> <p>Added information on reporting ancillary failures due to malloc failures at the end of section 4.3.8.4, “Guidelines for Allocating Memory”. (November 2006)</p> <p>Added subsection 4.7.2.1.1, “The blocktype Header and **previous”. Updated the reference to information on the show processes command and added an indexed reference to the show proc cpu command in subsection 4.7.2.14, “show processes memory Output”. (October 2006)</p>
Chapter 5, “Pools, Buffers, and Particles”	<p>Reorganized, updated, and merged sections throughout; added public buffer pool setup at system initialization time, and other missing pools, packet buffer, and particle API background. Added subsection 5.1.1, “Terminology”. Added subsection 5.2.8.1, “Pool Parameters in Dynamic Pools”, to describe setting the <code>pool_adjust()</code>, <code>maxcount</code>, <code>mincount</code>, and <code>permcount</code> parameters for managing dynamic pools. Added subsection 5.2.18, “Handle Throttling Conditions in a Pool with a Cache”, and moved interface throttling details there from the <code>pool_set_cache_threshold()</code> page of the Cisco IOS API Reference. Added use of the threshold callback vector to subsection 5.2.13, “Add a Pool Cache”. Added subsection 5.3.4, “Initialize System-wide Public Packet Buffer Pools”. Added subsection 5.3.10, “Reuse a Packet Buffer”. Moved section 5.5, “Useful Buffer Commands”, after particle pools because it describes commands related to both contiguous and particle-based buffers. Updated subsection 5.1.3, “Particle-based Buffers”, and subsection 5.4.1, “Particles: Overview”, for references to particle-unaware process-level forwarding code. (February 2007)</p>
Chapter 6, “Interfaces and Drivers”	<p>Added <code>HWSB_FAST</code> and <code>SWSB_FAST</code> definitions to subsection 6.5.3, “Subblock Modularity Data Structure Changes”. Updated section 6.1, “Introduction” and section 6.4, “Subblock Implementation Before Release 12.2S” to refer to section 6.5, “Subblock Implementation After Release 12.2S” (implemented in 12.2S and beyond). Updated section 6.6 to describe the subblock modularity changes as currently implemented. (January 2007)</p> <p>Added new section 6.16, “The Dev-Object Model”. (September 2006)</p>
Chapter 11, “Standard Libraries”	<p>Renamed chapter title from “ANSI C Library” to Chapter 11, “Standard Libraries”, to reflect including coverage of available POSIX standard function support (IEEE Std 1003.1-2001). Added Section 11.4, “Standard Library Support”, for brief background on the standards, links to available standards documentation, and to identify header file specification conventions for the standard library functions in different Cisco IOS release trains. (April 2007)</p>
Chapter 13, “Subsystems”	<p>In sections 13.2.3, “Subsystem Entry Point” and 13.2.4, “Subsystem Initialization Sequencing”, added how subsystems are discovered, and that subsystem init routines are run only once at system startup, but component initialization might be split across subsystem classes to allow multiple initialization stages (such as in EHSA implementations). Changed section 13.2.4.1 to 13.2.5, “Subsystem Classes”, to increase visibility. (February 2007)</p>

Chapter 14, “Registries and Services”	Added subsection 14.2.1, “System Registries”, with a table to summarize available system registries. Added subsection 14.2.2, “Registry Usage Dependencies”, to mention that the order in which registry callback functions are invoked depends on when they were added to the registry list, and to discourage but mention possible ways to handle timing dependencies among registry callbacks if necessary. (October 2006)
Chapter 16, “Timer Services”	Added new subsection 16.5.7, “Enhanced Timer Wheel (New in 12.4T)”. Added a description of timer wheels to section 16.5 and added a new subsection 16.5.1, “Timer Wheel Terms”. (April 2007)
Chapter 17, “Strings and Character Output”	Updated references to “ANSI C Library” chapter, which was renamed Chapter 11, “Standard Libraries” to include POSIX and ANSI C standard function support. (April 2007)
Chapter 20, “Debugging and Error Logging”	Changed title of section 20.6, “Debug Using Compile-Time Conditionals or Code Features” (was “Debug Using Compile-Time Conditionals”). Added subsection 20.6.2, “Unwedge an Input Queue Throttled Due to Buffer Leaks” to describe a new feature and CLI in 12.4(6)T that identifies and clears leaked packet buffers to unwedge an input interface that was throttled due to an out-of-buffers condition caused by a buffer memory leak. (January 2007) Added a new section 20.4, “How to Stop Buffered Debugging from Dropping buginfs()”. (December 2006)
Chapter 21, “Binary Trees”	Added <code>void *paramptr</code> to the input parameters for the <code>avl_walk()</code> function in subsection 21.3.1.3, “Traverse an AVL Tree”. (December 2006)
Chapter 25, “IP Services”	Added new section 25.4, “DNS”. (December 2006)
Chapter 27, “Command-Line Parser”	Due to Hot ICE updates, added information on replacing deprecated macros with the new <code>*_COMMAND</code> macros in section 27.5, “Hot ICE” (New in 12.5pi1, 12.4T, 12.2S). (January 2007) Moved and changed title of section 27.2.3.3, “NUMBER_STRICT Flag for GENERAL_NUMBER Macro (new in 12.2S and 12.4T)” to Section 27.3.3.1, “Octal Number Parsing in NUMBER and Other Related Macros”. To that section, added that NUMBER-related macros no longer support octal number parsing, except by explicit calls to octal-specific macros or the <code>GENERAL_NUMBER</code> macro with certain octal-processing flags set; added the new <code>OCTAL_NOT_STRICT</code> flag and noted that the <code>NUMBER_STRICT</code> flag previously added in 12.2s and 12.4 is deprecated in 12.4T. Added new <code>GENERAL_CHAR_NUMBER</code> macro and existing <code>CHAR_NUMBER</code> macro to the list in Table 27-4, “Macros for Parsing Numbers”. (October 2006)
Chapter 29, “MIB Infrastructure”	Added new subsection 29.2.1, “Interface Manager Infrastructure”. (August 2006)
Chapter 31, “AAA”	Added a new chapter on AAA (authentication, authorization and accounting) (New in 12.4(11)T). (March 2007)
Appendix A, “Writing Cisco IOS Code: Style Issues”	Added that the curly brace should be on the line following the function header under “Curly Braces” in section A.4.1, “Specific Code Formatting Issues”. (April 2007) Added section A.11.7, “Cross OS/Platform Driver Sharing”. (September 2006)
Appendix G, “Glossary”	Updated and added terms corresponding to pools, packet buffers, and particles chapter updates. (March 2007)

Changes in the Sixteenth Edition (July 2006)

The following changes have been made in the sixteenth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software.

Chapter	Change
Chapter 1, "Overview"	<p>Added new section 1.6, "How to Check software-d Threads". (July 2006)</p> <p>Added new section 1.5, "Determining Infrastructure Changes in Releases". (November 2005)</p>
Chapter 2, "System Initialization"	Added information about the new IOS Warm Reboot functionality to "IOS Warm Reboot," and its subsections. (November 2005)
Chapter 3, "Basic IOS Kernel Services"	<p>Expanded explanation of process_wait_for_event() in section 3.4.4.3.1. Removed example and added "No code outside of the IOS event-handler code should be calling lock_semaphore() and unlock_semaphore() directly" to section 3.4.3.7.1 "Locking and Unlocking an Atomic Lock: Example." (June 2006)</p> <p>Provided an improved description of thrashing in section 3.4.4.3.1. (December 2005)</p> <p>Added how to find the time spent between a message event posted and processed in subsection 3.4.3.9, see "process_send_message()". (October 2005)</p>
Chapter 4, "Memory Management"	<p>Added that non-dynamic memory chunks are not restricted to a maximum size of 64K in section 4.4.2, "Guidelines for Using the Chunk Manager". (May 2006)</p> <p>Added a new section 4.9, "Dynamic Bitlists". Added a note that sh mem 0x addr is now an internal command in section 4.7, "Memory Management Commands". (March 2006)</p> <p>Added description of memory lite command (new in 12.3T) to section 4.7.4 "Malloc-Lite Memory Commands." (February 2006)</p> <p>Added that the show chunk command is a hidden command in Table 4-9. (November 2005).</p>
Chapter 6, "Interfaces and Drivers"	Added new section 6.15 "Interface Locking Mechanism (new in 12.1)". (February 2006)
Chapter 8, "Interprocessor Communications (IPC) Services"	Added more commands and explanations for the output in section 8.23, "Useful Commands for Debugging an IPC Component". (November 2005)
Chapter 11, "ANSI C Library"	Added an explanation for the absence of floating point math.h functions in IOS in section 11.1, "Absence of Floating Point Functions". (December 2005)
Chapter 14, "Registries and Services"	<p>Added new section 14.16, "Manipulate FASTSTUB Services" (New in 12.2S, and 12.4T). (July 2006)</p> <p>Added a reference to the <i>Cisco IONization 101 Guidelines</i> for ION in subsection 14.5.2, "Placement of xxx_registry.o in Makefiles". (April 2006)</p> <p>Added registry information from the Cisco IOS Technical Note: Word of the Week (EDCS-184146) to subsection 14.1.3.1, "registry, service, service point". (February 2006)</p> <p>Added new subsection 14.1.6, "REMOTE Registries in IOS". (December 2005)</p>
Chapter 16, "Timer Services"	<p>Added new subsection 16.6.1, "The Effect of Changing a Watched Timer While Waiting for Events". (January 2006)</p> <p>Replaced the inaccurate text in subsection 16.3.4.1, "Determine the Current Time" with "obtain the number of milliseconds since system boot". (October 2005)</p>

Chapter 19, "Writing Cisco IOS Error Messages"	Added quotes around the header files in subsection 19.3.2.1, "Setting up #include's and #define's" and added some editorial changes and examples from Revision 14 of ENG-77462. (December 2005)
Chapter 20, "Debugging and Error Logging"	<p>Added subsection 20.1.2, "Event Tracing versus Debugging". (April 2006)</p> <p>Corrected the example of how to get a slot from the trace buffer in subsection 20.10.1.2.2, "Writing the Trace Hook Function". (March 2006)</p> <p>Added information on the debug sanity command to section 20.2.1 "Use Core Files to Debug CPU Exceptions." (February 2006)</p> <p>Added links to the memory leak detector tool and related documentation in section 20.7, "Links to Other Debugging Documentation". (October 2005)</p>
Chapter 23, "Switching"	Added a note that "This text is outdated, contact fc-uk@cisco.com for the updated CSSR (CEF Scalability and Selective Rewrite) information" to sections 23.1, 23.2, 23.3, and 23.4. (May 2006)
Chapter 25, "IP Services"	Added new section 25.3, "DHCP". (July 2006)
Chapter 27, "Command-Line Parser"	<p>Added new subsection 27.5.9, "Hot ICE Compliance Tooling". (July 2006)</p> <p>Changed the title of section 27.5, "IOS Management Robustness" to "Hot ICE" and added a reference to the new Hot ICE portal. Added an example with verification for each feature in subsections 27.5.2.9, 27.5.3.3, 27.4.3.4, 27.5.4.2, and 27.5.5.3, and updated sections 27.4, 27.5.2, 27.5.2.7, 27.5.3.2, 27.5.4.1, 27.5.5, and 27.5.6 for Hot ICE. (June 2006)</p> <p>Added new section 27.21 "Warning Against an Interactive CLI". (April 2006)</p> <p>Added new section 27.2.3.3 "NUMBER_STRICT Flag for GENERAL_NUMBER Macro (new in 12.2S and 12.4T)." (March 2006)</p> <p>Removed information on the <code>init_cfgdiff()</code> function from section 27.5.4.1, "The Config Rollback API" because the function is no longer needed before using the Config Rollback registry functions. Replaced section 27.18, "Tracking Down Parser EOLs with a Special 'Wumpus Hunt' Image" with section 27.19, "How to Find a Command's EOL". (February 2006)</p> <p>Removed section 27.15, "Adding Support for Show Defaults" because the show defaults and the show running-config defaults commands were removed from 12.4 and 12.4T (CSCds75679 is being reverted), and removed the reference to section 27.15 from the beginning of section 27.5.3. Added URL to command lookup website to section 27.10 "Useful Parser Commands." (January 2006)</p> <p>Added a new section 27.20, "NVGEN Enhancements". Added a new subsection 27.11.5, "Permanently Hidden Commands". Removed the link to the Wumpus Hunter's web page from section 27.18, "Tracking Down Parser EOLs with a Special "Wumpus Hunt" Image" because this website is for TAC's Wumpus Hunter effort, not the <code>wumpus_hunter</code> variable that is the topic of this section. (December 2005)</p> <p>Added a note on command dependencies and added a note on the exec mode enums in section 27.8, "Ordering Commands". Added a link to the Wumpus Hunters web page for more information on the Wumpus Hunt capability and related tools in section 27.18, "Tracking Down Parser EOLs with a Special "Wumpus Hunt" Image". Added subsection 27.3.1.4, "How to Avoid Collateral OBJ Damage in the Parse Trees". (November 2005)</p>
Chapter 30, "Security"	<p>Added new subsection 30.1.2, "printf() Pitfall". (April 2006)</p> <p>Added new section 30.1, "Secure Coding" with subsection 30.1.1, "How to Avoid Common Mistakes with Strings". (February 2006)</p>
Chapter 37, "Current Cisco IOS Initiatives"	Added links to online information on EDP and Phase Containment Training at the end of section 37.2, "Phase Containment Guidelines". (December 2005)

Appendix A, “Writing Cisco IOS Code: Style Issues”	Added the “Prototyping Functions That Take a Format String and Arguments” subsection. (July 2006) Added a subsection on enums to section A.6 “Coding for Reliability.” Added a link to a website with performance analysis tools and techniques in section A.8 “Coding for Performance.” Added that it is a convention in IOS to not use mixed case names in section A.5, “Variable and Storage Persistence, Scope, and Naming”. (February 2006) Replaced the colon with a semi-colon in the example of a correct return statement under “Spaces around Parentheses” in section A.4.1, “Specific Code Formatting Issues”. (November 2005)
Appendix B, “Cisco IOS Software Organization”	Added that questions can be directed to their respective areas, such as interest-parser@cisco.com or interest-os@cisco.com, and to software-d@cisco.com because code organization crosses a lot of boundaries. (April 2006)

Changes in the Fifteenth Edition (September 2005)

The following changes have been made in the fifteenth edition of the *Cisco IOS Programmer’s Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.4 of the Cisco IOS software.

Chapter	Change
Chapter 1, “Overview”	Rearranged sections. (September 2005)
Chapter 3, “Basic IOS Kernel Services”	Added information on how to see why a process doesn’t wake up after it has been sent messages, removed the statement that there is no way to see who’s holding the semaphore currently, and added that the show process event pid command prints information on the owner of the semaphore in section 3.4.5, “Useful Event Management Commands”. (July 2005) Updated the section 3.5, “Random Number Generation,” and the following subsections: 3.5.1, “Different Random Number Requirements,” 3.5.2, “Non-Cryptographic pRNG Functions,” 3.5.3, “IOS Universal Cryptographic pRNG Functions,” 3.5.4, “Crypto Image-Only IOS pRNG Functions,” 3.5.5, “Establishing an RNG Seed Context,” 3.5.6, “Obtaining a Random Number,” and 3.5.7, “Filling Memory with Random Numbers.” Added the new sections 3.5.8, “Secure Hash Algorithm with get_random_bytes() Function” and 3.5.9, “Further Information.” (April 2005)
Chapter 4, “Memory Management”	Corrected the description of show memory dead command output under subsection 4.7.2, “Memory Display Commands”. (September 2005) Added that you can use CHUNK_FLAGS_SMALLHEADER to avoid CPUHOG problems on chunk_free() if the chunk is dynamic and has a large number of elements to the note on CHUNK_FLAGS_SMALLHEADER in Table 4-8. (July 2005) Updated section 4.12, “ID Manager” with information on the new id_reseve() and id_table_set_no_randomize() functions in 12.4T. (May 2005)
Chapter 5, “Pools, Buffers, and Particles”	Added new subsection 5.9.4, “Packet Reparenting”. (July 2005)
Chapter 6, “Interfaces and Drivers”	Added that the default for swsb_link() is (sb_void_t) return_nothing and that a NULL check was added for sb->sb_ft->sb_link in section 6.5.4, “API Changes”. (September 2005) Added new section 6.6.4, “Iterate a List of Private IDBs Safely.” (March 2005)

Chapter 7, “Platform-Specific Support”	Added in section 7.6.2, “platform_buffer_init() Tasks”, that the <code>set_crashinfo_bufsize()</code> function is generally called by the platform code during system initialization, after the <code>platform_buffer_init()</code> function has been called. (May 2005)
Chapter 10, “Socket Interface”	Added information on how to list all open socket ports in section 10.2, “Useful Commands”. (July 2005) Added section 10.1, “SCTP Sockets API” (New in 12.4T). Added to relevant references “currently available with your CEC password at: http://cisco.safaribooksonline.com ”. (June 2005)
Chapter 11, “ANSI C Library”	Updated the current version of GCC used for IOS development to GCC 3.4 from <code>gcc.c2.95.3-p5</code> . (March 2005)
Chapter 13, “Subsystems”	Added information on runtime versus compile-time subsystems to section 13.1, “Overview: Subsystems”. Added new sections 13.2, “Runtime Subsystems” and 13.3, “Compile-time Subsystems”. (June 2005)
Chapter 14, “Registries and Services”	Replaced <code>void</code> with the <code>registry_status_e</code> return type in the <code>reg_add_name()</code> function for the <code>STUB_CHK</code> service in subsection 14.15.1, “Define a STUB_CHK Service”. (August 2005)
Chapter 17, “Strings and Character Output”	Added a note that routines must call <code>automore_disable()</code> before exiting in section 17.1.5, “Turn on Automatic “---MORE---” Processing”. (September 2005)
Chapter 19, “Writing Cisco IOS Error Messages”	Replaced section 19.4, “Testing the Error Message” because the <code>msgdef_serch.perl</code> script functionality is provided by <code>static_ios</code> . Added <code>msgdef_required_info()</code> to Table 19-1. (July 2005) Added new section 19.3.1.1 “msg_*.c files That Have Been Replaced by msg_*.rc Files.” (March 2005)
Chapter 20, “Debugging and Error Logging”	Added a new section 20.25, “How to Capture Console Output”. (September 2005) Added information on the UTF character encoding support to section 20.23, “TCL”. (July 2005) Added an example of how to enter GDB kernel debugging mode using <code>#g k</code> . Added links to information about GDB to section 20.2.3 “Debug with GDB.” Added that the Cisco GDB is an extension of the GNU/FSF GDB and added a link to the <i>Cisco Engineering Tools Guide</i> in section 20.2.1.2 “Analyze a Core File.” (March 2005)
Chapter 22, “Queues and Lists”	Added new section 22.8, “Iterate a List or Queue Shared with Interrupt Level Code.” (March 2005)

Chapter 27, "Command-Line Parser"	<p>Added some statements that are true for registry functions to section 27.3.9.6 "Registry". Replaced the information in subsections 27.2.1.3.1, "Converting from an Old CLI to a New CLI" and 27.2.1.3.2, "Deprecating a CLI" with a linked reference to ENG-31399, "Parser-Police Manifesto". (September 2005)</p> <p>Added section 27.5, "IOS Management Robustness" (New in 12.2S and 12.4T). Removed subsections "27.3.4 Data Structures", "27.3.5 Algorithmic Description", "27.3.6 Interface Design", "27.3.9 Sample PRC Implementation", "27.3.10 Development Unit Testing", and "27.3.11 Migrating Obsolete PRC to Current PRC" when subsection "27.5.2 PRC Phase III and Action Function Return Codes" was added. Added that section "27.15 Adding Support for Show Defaults" describes functionality that overlaps with that described in 27.5.3, but the exact same functionality is not available in the 12.2S train with a reference to see subsection 27.5.3, "Configuration Defaults Exposure" for information on the implementation in 12.2S. Removed section "27.16 Parser Syntax Checking" when subsection "27.5.5 Support for Syntax Checking" was added. Added the show parser dump exec command to Table 27-7. Added information on how the NO_OR_DEFAULT macro is used and revised text in subsection 27.3.7, "Negating Commands — the "No" and "Default" Keywords". (August 2005)</p> <p>Added information on the RES_* flags to subsection 27.2.1.1, "Parsing Config versus Commands". Renamed and renumbered subsection 27.10.1, "Exiting a Sub-mode" as subsection 27.12.4, "The exit Command from a Submode", and added new subsections 27.12.5, "Interface Submodes and Interface Ranges" and 27.12.2, "Exiting a Submode". (June 2005)</p> <p>Added KEYWORD_ID macro information to Table 27-1. Added new section 27.14, "CLI Safely Using Data Shared with Interrupt Level Code." (March 2005)</p>
Chapter 29, "MIB Infrastructure"	Removed the references to the <code>mib_nvwrite()</code> and <code>reg_invoke_write_mibs()</code> functions that are not in any major train. (September 2005)
Chapter 37, "Current Cisco IOS Initiatives"	Added new chapter. Added new section, "Phase Containment Guidelines". (April 2005)
Appendix A, "Writing Cisco IOS Code: Style Issues"	<p>Added information to "Function Prototypes" in section A.3.2, "Fifty Ways to Shoot Yourself in the Foot". Added new section A.2.7, "A.2.7 '<\$paratext[AppSectionsub]>'" Provided more explanation and example code to the Switch Statements and Default Cases portion of section A.6, "Coding for Reliability." (September 2005)</p> <p>Added note that "Preprocessor symbols defined in header files included by more than one or two C files should have at least a two character prefix." to Appendix A.5 "Variable and Storage Persistence, Scope, and Naming." (August 2005)</p> <p>Added a note to contact <code>ios-serviceability@cisco.com</code> for any questions on coding for serviceability under section Appendix A.1.3, "A.1.3 '<\$paratext[AppSectionsub]>'". (June 2005)</p>

Changes in the Fourteenth Edition (February 2005)

The following changes have been made in the fourteenth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.3 of the Cisco IOS software.

Chapter	Change
Chapter 1, "Overview"	Updated references to the Network Interface Drivers documentation. (November 2004)

Chapter 2, "System Initialization"	Added new section "Retrieving System Information". (September 2004)
Chapter 3, "Basic IOS Kernel Services"	<p>Added section 3.3.6, "How to Preempt IOS Processes" (New in 12.2S and the MIPS platforms in 12.0S per the "Commits-note" enclosure of CSCec04080). (February 2005)</p> <p>Added a note on managed semaphore usage to section 3.4.3.6.1 "Managed Semaphore: Definition." Added a note on atomic lock usage to section 3.4.3.7 "Atomic Locks." (January 2005)</p> <p>Added a new section 3.4.5 "Useful Event Management Commands". Corrected statements in sections 3.4.3.9 for "process_get_message()" and "process_get_watched_message()". (November 2004)</p>
Chapter 4, "Memory Management"	<p>Added a note and more details regarding dead processes to section 4.7.2.3, "show memory dead Output." (February 2005)</p> <p>Added a new section 4.7.5, "How to Check Blocks and Buffers". (November 2004)</p> <p>Added a note to the description of <code>CHUNK_FLAGS_SMALLHEADER</code> in Table 4-8 "Chunk Pool Flags." (August 2004)</p>
Chapter 6, "Interfaces and Drivers"	<p>Added a new section 6.14 "nextsub Subinterface List (new in 12.2T)." (per CSCdu44151) (new in 12.2T) (January 2005)</p> <p>Updated a reference to the Network Interface Drivers documentation. (November 2004)</p> <p>Added a new section 6.13 "The linktype enum". (new to 12.3 and 12.2S) (September 2004)</p>
Chapter 7, "Platform-Specific Support"	<p>Added subsection, "List of Platform Code Terms" to distinguish platform-specific from platform-dependent, platform-generic, and platform-independent. Added subsection 7.8.6.2, "PA Function Calls". (November 2004)</p> <p>Added section 7.17, "How to Request Platform-Specific Action from Platform-Independent Code". (September 2004)</p>
Chapter 9, "File System,"	<p>Added new sections 9.3.1.6 "Example 4 - Accessing the File System," 9.3.1.9 "Example 5 - Reading a File," 9.3.1.8 "Example 6 - Writing to a File," 9.3.1.9 "Example 7 - Seeking a Position within a File," 9.3.1.10 "Example 8 - Renaming a File, 9.3.1.11 "Example 9 - Removing a File," 9.3.1.12 "Example 10 - Making and Removing a Directory," 9.3.1.13 "Example 11 - Collect File System Statistics," 9.3.1.14 "Example 12 - IOCTL Function Support and Getting Next Directory Entry," and 9.3.1.15 "Example 13 - Closing All Files." Revised code examples in sections 9.7.1 "Setting a Value to a Persistent Variable," 9.7.3 "Incrementing the Value of a Persistent Variable," and 9.7.4 "Deleting a Persistent Variable from nvram:persistent-data." Added references to 9.8 "References" for further reading. (September 2004)</p> <p>Added section 9.6, "NVRAM API". (August 2004)</p>
Chapter 11, "ANSI C Library"	Added links to GCC documentation. (November 2004)
Chapter 14, "Registries and Services"	<p>Added information on how to add and remove registries after the Registry Redefine Project in Step 2b, 14.5 "Steps to Create and Use a Registry" (New in 12.2S and 12.3T PI06). Added a note that only one callback function can be registered at a time for each case in CASE and RETVAL registries to sections 14.10, "Manipulate CASE Services" and 14.11, "Manipulate RETVAL Services." (January 2005)</p> <p>Added step 2b to section 14.5 "Steps to Create and Use a Registry" which discusses the <code>FILES_REG</code> variable. (September 2004)</p>

Chapter 15, “Time-of-Day Services”	Added information about the system clock to section 15.1.3, “System Clock: Description.” Added information about <code>clock_get_time()</code> to section 15.1.5, “Network Time Protocol.” Added a new section 15.1.6, “System Clock Changes.” (February 2005)
Chapter 17, “Strings and Character Output”	Added that “Furthermore, <code>printf()</code> cannot be used when interrupts are disabled.” to section 17.1.2.1 “Differences between <code>buginf()</code> and <code>printf()</code> .” (February 2005)
Chapter 19, “Writing Cisco IOS Error Messages”	Corrected the syntax in the example of <code>msgdef_limit()</code> at the end of subsection 19.3.2.3. (December 2004)
Chapter 20, “Debugging and Error Logging”	Added that “Furthermore, <code>printf()</code> cannot be used when interrupts are disabled.” to section 20.3.1.1 “Differences between <code>buginf()</code> and <code>printf()</code> .” (February 2005) Updated the list of URLs in section 20.7, “Links to Other Debugging Documentation”. (December 2004)
Chapter 21, “Binary Trees”	Improved the function descriptions in sections 21.3.2.1, “Using WAVL Data Structures and Defining Necessary Routines,” 21.3.2.2, “Initialize a Wrapped AVL Tree,” 21.3.2.3, “Insert a Node into a Wrapped AVL Tree,” 21.3.2.4, “Traverse a Wrapped AVL Tree,” and 21.3.2.5, “Search a Wrapped AVL Tree.” (January 2005)
Chapter 23, “Switching”	Added section 23.6, “L2VPN Platform API” (New in 12.2S RLS7, except the <code>atom_smgr</code> APIs are in 12.0S). (February 2005) Updated references to the Network Interface Drivers documentation. (November 2004)
Chapter 27, “Command-Line Parser”	Revised the note and added a new note to the section 27.3.7, “Negating Commands — the “No” and “Default” Keywords.” (February 2005) Added the new section 27.3.3.3, “Example: Parse a Number from a Discontiguous Range.” Updated the section 27.3.9, “How to Block Unsupported Commands.” (January 2005) Clarified a statement about included files at the end of subsection 27.3.1.2 “Parser Chains”. (December 2004) Added the new section 27.18 “Tracking Down Parser EOLs with a Special “Wumpus Hunt” Image”. (November 2004) Added the new section 27.11 “Guidelines for Internal and Hidden Commands and More.” (October 2004) Changed <code>{PARSE_LIST_END, NULL}</code> to <code>(parser_chain_id(LIST_END)), NULL}</code> in section 27.8.1 “Linking Commands”, section 27.9.4.1 “Example: Link Commands to a Link Point” and section 27.9.5.1 “Example: Create Link Exit Points”. Added how to uniquely identify the link point in 12.3 and later releases to section 27.9.1.1 “Example: Create a Link Point”. These are all SingleSource-related changes to the parser. Added the new sections 27.3.7 “Negating Commands — the “No” and “Default” Keywords,” 27.3.7.1 ““No” versus “Default”,” 27.3.7.2 “Handling the Negative Forms of Commands,” and 27.3.7.4 “Parameter Elision in Negative Forms - the NOPREFIX Macro.” (September 2004)
Chapter 28, “Writing, Testing, and Publishing MIBs”	Corrected links to RFCs. (October 2004)
Chapter 30, “Security”	Added section 30.2.2, “How to Use Private VRFs to Isolate Interfaces”. Added a new section 30.2 “Integer Manipulation Vulnerabilities.” (October 2004)

Appendix A, "Writing Cisco IOS Code: Style Issues"	Added a new section titled, A.2.1.2, "Guidelines for Using the ## Operator." (February 2005)
	Added a new section titled, A.11, "Coding for Multiple Operating Environments." Added "Avoiding Unlabeled Constants" information to section A.2.1 "Do Not Abuse the Pre-Processor." Added that spaces should be used around operators to section A.4.1 "Specific Code Formatting Issues." (December 2004)
	Added a new section titled, A.10, "Coding for Security." (November 2004)
	Added a new section titled, "A.7 Coding for Usability" and added information about conditional compilation to "A.2.1.1 Avoid Conditional Compilation". (October 2004)
Appendix C, "CPU Profiling"	Added new section C.2.5.1 "Notes for Dealing with CPUHOGs." (January 2005)
Appendix F, "Embedded Documentation"	Added a new appendix with documentation on DocX. (January 2005)
Appendix G, "Glossary"	Added Platform-dependent/generic/independent/specific Code, MCEF, NRP, NSP, PAM MBOX, PAM MBOX interface, VRF, and VRFIfying. Added definitions of alignment correction, bus error, and spurious accesses. Updated references to the Network Interface Drivers documentation. (November 2004)

Changes in the Thirteenth Edition (July 2004)

The following changes have been made in the thirteenth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.3 of the Cisco IOS software.

Chapter	Change
Chapter 1, "Overview"	Added new section 1.7 "Migrating to ION". (June 2004)
Chapter 2, "System Initialization"	Fine-tuned explanation of how ROMMON boots an image in section "Detailed Example of how the ROMMON Boots an Image". Added new section "IOS Warm Upgrade". (April 2004) Added new section "Out of Sequence Response Code". (March 2004) Added more information on using interrupt levels in Cisco IOS in section, "Setting the Interrupt Levels". Per CSCed24017, replaced the <code>interrupt.h</code> include file with <code>COMP_INC(kernel, ios_interrupts.h)</code> for the <code>get_interrupt_level()</code> , <code>raise_interrupt_level()</code> , <code>reset_interrupt_level()</code> , and <code>set_interrupt_level()</code> functions in section due to the componentization effort. (December 2003) Added new section "Detailed Example of how the ROMMON Boots an Image". (November 2003) Added a note on Cisco IOS system initialization questions. (October 2003)
Chapter 3, "Basic IOS Kernel Services"	Updated section 3.2.1 "Cisco IOS Task States" with further explanation of the idle (or ready-to-run) state. Updated section 3.2.6 "Important CLI Commands" with further explanation of the max-task-time 200 CLI. Added scheduler process-watchdog command. Updated Table 3-2 explaining that <code>process_may_suspend()</code> has changed to allow scheduler support for guaranteed quantum for Normal/Low priorities. Updated section 3.4.3.2 "Event Operations" with further explanation of the scheduling of one or all IOS tasks. (February 2004) Added new section 3.4.3.8 "Read/Write Locks"(New in 12.3T). (December 2003) Added a note on kernel services and scheduler development questions. (October 2003)

Chapter 4, "Memory Management"	<p>Added more details on the chunk_free() function to section 4.4.4 "Allocate and Return a Memory Chunk Element". (July 2004)</p> <p>Added a new section 4.4.3.2.1 "A Description of Chunk Sibling Chaining". Added the mem-busters@cisco.com and interest-mem@cisco.com aliases to the note on memory management development questions. (June 2004)</p> <p>Added the URL for the show processes command documentation to section 4.7.2.14 "show processes memory Output". Updated section 4.7 "Memory Management Commands" to provide an exhaustive list of show commands and their corresponding output. Added the "4.12 ID Manager" section (New version in 12.3T). (May 2004)</p> <p>Added the "4.8 Dynamic Bitfield Management" section. (February 2004)</p> <p>Added the new API functions from the Transient Memory Allocation feature to subsection 4.1.1 "List of Memory Management Functions." Added a new entry: MEMPOOL_CLASS_TRANSIENT to Table 4-6 in subsection 4.3.5.3 "List of Common Memory Pool Classes." Added two API functions to Table 4-7 in subsection 4.3.8.3 "Comparison of Memory Allocation Functions." Added the CHUNK_FLAGS_TRANSIENT definition to Table 4-8 in subsection 4.4.3 "Create a Memory Chunk." Added the new section 4.6 "Transient Memory Allocation (New in 12.2S)." Added a note on memory management development questions. (October 2003)</p>
Chapter 5, "Pools, Buffers, and Particles"	<p>Added section 5.9.3.1, "Specifying a Clone Queue When Cloning Packets." (June 2004)</p> <p>Added section 5.5, "Useful Buffer Commands". (March 2004)</p> <p>Added a note on pool, buffer, and particle development questions. (October 2003)</p>
Chapter 6, "Interfaces and Drivers"	<p>Added a new section 6.3.7 "Address Filter Function Vectors." (New for 12.3, 12.2, 12.2S, and 12.1E) (June 2004)</p> <p>Added a new section 6.2.7 "ATM IDB Recycling." (New for 12.0S) (April 2004)</p> <p>Added "6.2.8 IDB Protocol Counter API" (New in 12.2S). (March 2004)</p> <p>Added a note on Cisco IOS interfaces and drivers development questions. Added a warning to subsection 6.4.3 "Add an IDB Subblock," that because there are no checks to prevent a single subblock from appearing in multiple IDBs' lists, you must make sure that you do not add the same subblock to different IDBs. (October 2003)</p>
Chapter 7, "Platform-Specific Support"	<p>Added information on indicating support for NVRAM variables in a new subsection titled, 7.7.4 "platform_nvvar_support()". (December 2003)</p> <p>Added "For platforms that have async serial interfaces and use slotted architecture: hwidb->slotunit = hwidb->unit = tty->ttynum" to section 7.15.9.1 "Other usages of hwidb->subunit" to help indicate that async interfaces are "oddly" behaved. Added information on when the <code>platform_interface_init()</code> function calls <code>reg_add_subsys_init_class()</code> in relation to the sequence of subsystem initialization and the driver initialization routine at the end of "7.8.2 <code>platform_interface_init()</code> Tasks". (November 2003)</p> <p>Added a note on platform-specific support development questions. (October 2003)</p>
Chapter 8, "Interprocessor Communications (IPC) Services"	Added a note on Cisco IOS IPC development questions. (October 2003)
Chapter 9, "File System"	Added a note on file system development questions. (October 2003)
Chapter 10, "Socket Interface"	<p>Added socket_get_max_sockets() and socket_set_max_sockets(), which are new in 12.3PI5). (December 2003)</p> <p>Added a note on socket interface development questions. (October 2003)</p>

Chapter 11, "ANSI C Library "	Added a note on ANSI C development questions. (October 2003)
Chapter 12, "CNS"	Added a note on CNS development questions. (October 2003)
Chapter 13, "Subsystems"	Updated section 13.1.1 "Subsystem Classes" with latest subsystem classes and a note on a good way to tell exactly what subsystem classes are supported. (December 2003) Added a note on Cisco IOS subsystems development questions. (October 2003)
Chapter 14, "Registries and Services"	Updated <code>reg_delete_name()</code> to correct parameters for service type in section 14.10.1 "Define a CASE Service", section 14.12.1 "Define a FASTCASE Service", section 14.13.1 "Define a LOOP Service", section 14.14.1 "Define a STUB Service", section 14.15.1 "Define a STUB_CHK Service" and section 14.18.1.9 "Delete the Callback Function from the Main List". Added a new subsection 14.1.5 "Default Functions." Added subsections 14.7.2 "LIST Service's Default Function," 14.8.2 "ILIST Service's Default Function," 14.9.2 "PID_LIST Service's Default Function," 14.10.2 "CASE Service's Default Function," 14.11.3 "retval Service's Default Function," 14.12.2 "FASTCASE Service's Default Function," 14.13.2 "LOOP Service's Default Function," 14.14.2 "STUB Service's Default Function," 14.15.2 "STUB_CHK Service's Default Function," and 14.17.2 "VALUE Service's Default Function." Inserted descriptions of service functionality for CASE_LOOP and CASE_LIST to subsection 14.6.1.2 "Comparison of Service Type Functionality." (February 2004) Added a note on registries and services development questions. In 12.2, the <code>reg_used_name()</code> functions are only provided for the CASE registries. In 12.2T, the <code>reg_used_name()</code> functions are also provided for STUB and LIST registries. Added subsections 14.7.1.4 "Example: Inquire on Registrations of a LIST Service", and 14.14.1.4 "Example: Inquire on Registrations of a STUB Service". Reworded the subsection 14.10.1.3 "Example: Inquire on Registrations of a Particular CASE Value". (October 2003)
Chapter 15, "Time-of-Day Services"	Added a note on Cisco IOS time-of-day services development questions. (October 2003)
Chapter 16, "Timer Services"	Added a reference to how parent timers can have context in subsection 16.4.10, "Modify the Timer Context", and added when the router can complain about an uninitialized timer in subsection 16.4.14, "Stop a Managed Timer". (November 2003) Added a note on Cisco IOS timer services development questions. (October 2003)
Chapter 17, "Strings and Character Output"	Updated Table 17-6 "Examples of Formatting Timestamps" to include the variants for other formats. (March 2004) Added a note on strings and character output development questions. (October 2003)
Chapter 18, "Exception Handling"	Added a note on Cisco IOS exception handling questions. (October 2003)
Chapter 19, "Writing Cisco IOS Error Messages"	Updated "Submitting Error Messages for Review", "Creating an Error Message Explanation", and "Specifying a Recommended Action" per December 2003 updates to ENG-77462. (January 2004) Added a note on error message development questions. (October 2003)

Chapter 20, “Debugging and Error Logging”	<p>Added paragraph on what happens when a large number of messages are generated such that the console output is flushed to section 20.3 “Debug with buginf() and the debug Command”. Added that <code>buginf()</code> does not suspend to section 20.3.1.1 “Differences between <code>buginf()</code> and <code>printf()</code>”. Added a new section 20.22 “Memory Traceback Recording” (<i>New in 12.3T</i>). (April 2004)</p> <p>Added a note in section 20.2.1 “Use Core Files to Debug CPU Exceptions” that as of Release 12.2S and 12.3, the core files generated from IOS will now carry a timestamp. (March 2004)</p> <p>Added a note on debugging and error logging development questions. (October 2003)</p>
Chapter 21, “Binary Trees”	<p>Updated the current use of a Cisco IOS radix tree in “21.1.3 Radix Trees”. (November 2003)</p> <p>Added a note on binary tree development questions. Added new section 21.3.3 “Manipulate Threaded AVL Trees”. (October 2003)</p>
Chapter 22, “Queues and Lists”	<p>Added new section 22.10 “Index Tables”. (<i>New in 12.2T</i>) (January 2004)</p> <p>Added a note on queue and list development questions. (October 2003)</p>
Chapter 23, “Switching”	<p>Added “23.5 Hardware Session and L2 Hardware Switching” (<i>New in 12.3T</i>). (April 2004)</p> <p>Corrected the command used to view private buffer pools to show buffers (the show buffers interface command does not exist) in section 23.1.1, “Low-End and Mid-Range Systems”. (November 2003)</p> <p>Added a note on switching development questions. (October 2003)</p>
Chapter 24, “High Availability (HA)”	Added a note on Cisco IOS HA development questions. (October 2003)
Chapter 25, “IP Services”	Added a new chapter for IOS IP Services. Added a new section, “BEEP” (<i>New in 12.2S RLS7</i>). (July 2004)
Chapter 26, “Porting Cisco IOS Software to a New Platform”	Added a note on Cisco IOS software porting questions. (October 2003)
Chapter 27, “Command-Line Parser”	<p>Added section 27.3.11, “Configuration Editor in IOS?” (July 2004)</p> <p>Added the URL for more information on the privilege configuration command to section 27.3.2 “Parse a Keyword Token”. Referred to EDCS-186494 in section 27.4 “Adding Parser Return Codes” for more information on a step by step procedure for migrating existing IOS Features to parser return codes. (June 2004)</p> <p>Added new section 27.7.3 “Important Notes on Data Variables”. A new <code>CASE_LOOP</code> registry service, <code>config_is_cli_disallowed()</code>, has been added to 12.0S, 12.2S, and 12.3T. Information about it has been added to subsection 27.3.9 “How to Block Unsupported Commands.” (March 2004)</p> <p>Added <code>PRIV_NULL</code> and <code>PRIV_OPR</code> to Table 27-2 “Flags for Specifying Privilege Level When Parsing Keywords”. Added <code>PRIV_DISTILLED</code>, <code>PRIV_INTERACTIVE</code>, <code>DEFAULT PRIV</code>, and <code>PRIV_NO_SYNALC</code> to Table 27-3 “Flags for Specifying Other Options When Parsing Keywords”. Added subsection 27.2.1.3.2, “Deprecating a CLI”. (February 2004)</p> <p>Added new section 27.8 “Ordering Commands.” (January 2004)</p> <p>Added a note on CLI output development questions. (December 2003)</p>
Chapter 29, “MIB Infrastructure”	Added a note on Cisco IOS MIB infrastructure questions. (October 2003)
Chapter 30, “Security”	Added a note on Cisco IOS security development questions. (October 2003)

Chapter 32, "Scalable Process Implementation"	Added a note on Cisco IOS scalable process questions. (October 2003)
Chapter 34, "Verifying Cisco IOS Modular Images"	Added a note on Cisco IOS modular image questions. (October 2003)
Chapter 35, "Writing DDTs Release-note Enclosures"	Added a note on Cisco IOS Release-note enclosure questions. (October 2003)
Appendix A, "Writing Cisco IOS Code: Style Issues"	Added the "A.2.1 Do Not Abuse the Pre-Processor" subsection and moved "Do Not Use Conditional Compilation for Platform-Specific Code", "Header Files", and "Enumerated Types and #defines" (with the first example replaced) into it. Also, added information on implicit conversions to "Conversion from Signed to Unsigned Types" and added a note on code reliability to "A.8 Coding for Performance". (June 2004) Updated the information about switch statements and default cases. Added a note about the use of break in cases in section A.4 "Presentation of the Cisco IOS Source Code." Corrected a statement about checking for NULL pointers in section A.6 "Coding for Reliability." (April 2004) Added a note on Cisco IOS code style questions. Updated the edition in the reference to <i>Introduction to Algorithms</i> in section A.8.1, "Performance of Algorithms and Data Structures". (October 2003)
Appendix B, "Cisco IOS Software Organization"	Added a note on Cisco IOS code organization questions. Rewrote section B.5 and changed title of section B.5 "Alternatives to the Default CPU-Specific Object Directories". (October 2003)
Appendix C, "CPU Profiling"	Added a note on Cisco IOS CPU profiling questions. (October 2003)
Appendix D, "Older Version of the Scheduler"	Added a note on Cisco IOS scheduler questions. (October 2003)
Appendix E, "Branch Integration & Sync Processes"	Added a note on Cisco IOS branch integration and sync process questions. Updated Figure E-1, Figure E-2, Figure E-9, Figure E-12, Figure E-13, Figure E-14, Figure E-15, Figure E-16 and Figure E-17 with more accurate diagrams. (October 2003)
Glossary	Added the definition of a Cisco IOS radix tree. Added a linked URL to the "Engineering Education Technical Glossary". (November 2003)

Changes in the Twelfth Edition (September 2003)

The following changes have been made in the twelfth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.3 of the Cisco IOS software.

Chapter	Change
Chapter 1, "Overview"	Added new section 1.3.4 "IDB Data Structure Shrinking" and section 1.3.5 "IDB Subblock Modularity". Added new section 1.3.1.8 "File System", section 1.3.1.10 "CNS", and section 1.3.3.4 "High Availability (HA)". Removed ISR from section 1.3.6 "Other Scalability Changes". This is not in use. Added new section 1.3.6 "Other Useful Information" and section 1.3.7 "Appendices". (September 2003)
Chapter 2, "System Initialization"	Added new section "Setting the Interrupt Levels" which had been inadvertently omitted. (August 2003) Added the "IOS Warm Reboot" section (New in 12.3T). (June 2003)

Chapter 3, "Basic IOS Kernel Services"	<p>Added information about <code>process_max_time()</code> defaults in the 3.2.6 "Important CLI Commands" section. Added information on thrashing to the 3.4.4.3.1 subsection. (June 2003)</p> <p>Added a new paragraph to section 3.3.3.2 "Creating a Cisco IOS Task" regarding the fixed stack size of a process and because of this, avoiding large automatic variables. (May 2003)</p> <p>Added new section 3.5 "Random Number Generation". (March 2003)</p>
Chapter 4, "Memory Management"	<p>Added the "4.4.3.2 Dynamic Chunk Sibling Creation and Destroy" subsection. (September 2003)</p> <p>Updated Table 4-9 in section 4.7.2 "Useful Memory Commands" with show processes memory row. (August 2003)</p> <p>Added URL for more show commands in section 4.7.2 "Useful Memory Commands". (July 2003)</p> <p>Added memory malloc-lite-use-malloc and memory try-malloc-lite to the "4.7.3 Memory Validation Commands" section. (June 2003)</p> <p>Added in section 4.10, "Virtual Memory", that the VM documentation remains valid for Release 12.3 because VM has not changed significantly since Release 12.0 when the documentation was written. Added in subsection 4.10.5.1, "Virtual Addresses vs. Physical Addresses", that in Cisco IOS, both the TEXT and LOCAL areas are virtual for platforms that support VM, and VM is used primarily to page out and page in pages in the TEXT segment to save space. (May 2003)</p> <p>Added section 4.11, "Caching Issues". Added a description of dead memory in the "4.7.2 Useful Memory Commands" section. (April 2003)</p>
Chapter 5, "Pools, Buffers, and Particles"	<p>Added subsections 5.3.11 "Prune a Packet Buffer Pool" and 5.4.9 "Prune a Public Particle Pool" that describe the use of the <code>POOL_PRUNE_TIME</code> constant. (September 2003)</p> <p>Added new section 5.5.13 "Useful Commands Regarding Buffer Pools". (July 2003)</p> <p>Added new section 5.9.3 "Particle Clones" which had been inadvertently omitted. (May 2003)</p> <p>Added that <code>datagram_done()</code> should be used instead of <code>retbuffer()</code> in section 5.3.9 "Return a Packet Buffer to a Pool". (February 2003)</p>

Chapter 6, "Interfaces and Drivers"	<p>Changed the name of subsection 6.5.2 from "Subblock Modularity Changes" to "Subblock Modularity/Scalability Changes;" added information describing three new entries to the subblock's function table; the bullet item that begins: "Remove the link and unlink function table entries..." was changed to: "Remove the enqueue and unlink function table entries...;" a new bullet was added: "Add a new sb_link function table entry after the pointer to the list header to specify whether to link a subblock in its owning list IDB-sorted order." In subsection 6.5.3 "Subblock Modularity Data Structure Changes" <code>sb_void_t sb_link;</code>, was added to the <code>sb_ft</code> structure. The paragraph beneath this structure was also updated to reflect the change. The "typical subblock function table" was updated with <code>hwsb_link;</code>. The <code>#define HWSB_BASE_LIST</code> was updated to: <code>#define HWSB_BASE_LIST sb_t hwsb_base; sb_t *sblist_next; sb_t *sblist_prev;</code>. In subsection 6.5.4 "API Changes" an extra bullet was added describing the standard destroy functions <code>hwsb_destroy()</code> and <code>swsb_destroy()</code> that have been added to perform the combined action of deleting and freeing a subblock. In subsection 6.6.5 "Adding a new subblock," the <code>my_sub_ft</code> function tables and the <code>my_sub_destroy()</code> function were updated. In section 6.5.7 "Guidelines to Convert to New Subblock Implementation," added that because of the modularity/scalability changes, the enum that defined IDs for all subblocks has been removed and the subblock model changed so that a subblock's ID is not assigned until the subblock is actually created. (September 2003)</p> <p>Added new section 6.12 "Protecting an IDB from the User". (June 2003)</p> <p>Added subsection 6.3.4.3, "Cisco IOS Virtual Functions versus IDB Function Vectors". (May 2003)</p> <p>Added the "6.11 Getting and Setting IDB Fields" section on using the list parameter with the <code>idb_hw_state_config()</code> function for getting or setting the state, configuration, or statistics fields for IDBs. (March 2003)</p>
Chapter 7, "Platform-Specific Support"	Added the <code>PLATFORM_VALUE_LOG_MAX_MESSAGES</code> value to Table 7-2. (May 2003)
Chapter 8, "Interprocessor Communications (IPC) Services"	Added new section 8.24 "IPC Master". (April 2003)
Chapter 9, "File System"	Added a reference to EDCS-280183 for information on file open flags in section 9.4.3. (May 2003).
Chapter 11, "ANSI C Library"	Replaced information on the version of C used in IOS with the currently supported C compiler version. (March 2003)
Chapter 12, "CNS"	Added information about the Namespace Mapper (NSM) in "12.1.3 Event Agent Services". (June 2003)
Chapter 13, "Subsystems"	Added that by default sequencing within a particular subsystem class is caused by link order in section 13.4.1, "Subsystem Property Definitions". (June 2003)
Chapter 14, "Registries and Services"	<p>Added new sections 14.1.4.1 "What Are the Two Common Uses of Registries?", 14.1.4.2 "Two Other Ways to Think of Registries" and 14.1.4.3 "Words of Guidance When Adding New Registry Entries". Added new section 14.18 "Manipulate CASE_LIST/CASE_LOOP Services". (April 2003)</p> <p>Added new section 14.1.4 "Common Uses Of Registries". (March 2003)</p> <p>Added a note on when to add paths in the <code>FILES_REG</code> variable in the "Steps to Create and Use a Registry" section. (February 2003)</p>
Chapter 16, "Timer Services"	Added information on timers and the time offset value when it is within the granularity range to "16.6 Choose Which Type of Timer to Use". (September 2003)

Chapter 19, "Writing Cisco IOS Error Messages"	Revised entire chapter with updated procedures and current examples, per EDCS 77462 Rev. C. Added subsection 19.5.3 "Use <code>errmsg_ext()</code> in Remote Registry Calls" for the new <code>errmsg_ext()</code> function (New in 12.2(105)S). (July 2003) Added a note on the <code>PLATFORM_VALUE_LOG_MAX_MESSAGES</code> value to subsection 19.3.2. (May 2003) Added subsection 19.4.1, "Testing Error Message Format". (April 2003)
Chapter 20, "Debugging and Error Logging"	Added note regarding syslog and error messages of severity level equal or higher than <code>log_reload</code> in section 20.2 "The Debug Facility and Exceptions". (September 2003) Added new section 20.23 "TCL" (new in 12.3T). Added new section 20.24 "Embedded Syslog Manager" (new in 12.3T). Updated information on <code>EVENT_TRACE_BASIC_INSTANCE</code> in the "20.10.1.2 Framework of ipc_trace.c" subsection. (July 2003) Added information on redirecting the <code>show</code> command output for debugging purposes in the "Debugging and Error Logging Facilities Overview" section. (March 2003).
Chapter 21, "Binary Trees"	Added section 21.5 "String Database for Fast Lookup". (July 2003)
Chapter 22, "Queues and Lists"	Replaced "void" with "boolean" for the following functions in section 22.2 "Manipulate Queues" that were changed in 12.2T: <code>queue_init()</code> , <code>enqueue()</code> , <code>requeue()</code> , <code>insqueue()</code> , <code>unqueue()</code> , <code>p_enqueue()</code> , <code>p_requeue()</code> , <code>p_unqueue()</code> and <code>p_unqueueunext()</code> - because of an implementation change in 12.2T and above. (August 2003) Added new section 22.7 "Other Doubly Linked List Functions" as part of doubly linked list functionality. (May 2003)
Chapter 23, "Switching"	Added URL for more <code>show</code> commands in section 23.1.1, "Low-End and Mid-Range Systems". (July 2003)

Chapter 27, "Command-Line Parser"	<p>Added notes about using <code>reg_invoke_isc5rsp()</code> and <code>reg_invoke_is_keyword_supported()</code> and also added new information about using <code>reg_invoke_is_qos_int_cli_disabled()</code> to the subsection 27.3.9 "How to Block Unsupported Commands." Added note about defining <code>PARSER_DEBUG_LINKS</code> (New in 12.2S) in order to make the show parser links command available in subsection 27.9.3 "Display Registered Link Points." Added new section 27.9 "Guidelines for Hidden Commands". Added new section 27.12 "Useful Exit Functions and Submode Attributes". (September 2003)</p> <p>Added the "27.16 Adding New Interface and Controller Types" section (New in 12.3T). Added a note on help string format in "27.2.2 Transition Structure". Added the "27.3.1.2 Parser Chains" subsection. Added the "27.17 ICD (Intelligent Config Diff)" section (New 12.3T). Added the "27.18 IOS Config Rollback" section (New 12.3T). (July 2003)</p> <p>Added the "27.15 Parser Syntax Checking" section (New in 12.3). Added the "27.15 Adding Support for Show Defaults" section. Added the "27.3.11 Migrationg Obsolete PRC to Current PRC" subsection and replaced the obsolete PRC macros in the "27.4 Adding Parser Return Codes" section. (June 2003)</p> <p>Added subsection 27.3.9, "How to Block Unsupported Commands". (April 2003)</p> <p>Added "Useful Parser Commands" section. Added references to ENG-91860 and ENG-73672 in the section and to ENG-91860 in the "Debugging Parser Ambiguity" section. Added the "The "no" Prefix Argument" and "Setting Defaults and the "default" Prefix Argument" subsections. Added a note on the parser-questions and parser-police aliases and the parser-police processes in the "Overview: Parser" section. Added a note on where to introduce an NVGEN parser node in the "Nonvolatile Output Generation" subsection. Added note regarding the <cr> option and the EOL node in the "Traversing the Parse Tree" section. Added the "Parsing Config versus Commands" subsection. Added the "Maintaining Backward Compatibility" subsection. (March 2003)</p>
Appendix A, "Writing Cisco IOS Code: Style Issues"	<p>Removed unneeded text about emacs in section A.3, "Using C in the Cisco IOS Source Code". (June 2003)</p> <p>Added new section A.2.6 "A.2.6 '<\$paratext[AppSectionSub]>'>" regarding the fixed stack size of a process and because of this, avoiding large automatic variables. (May 2003)</p> <p>Added that %p is the correct format specifier for a pointer under "Format Specifier for a Pointer". (February 2003)</p>
Appendix B, "Cisco IOS Software Organization"	<p>Added new section B.5 "What is the Purpose of the obj-4k Directory?". Added "For the following files, please check the ..//cisco.comp/track file for the full path name." to section B.1.1 "B.1.1 '<\$paratext[AppSectionSub]>'>" (August 2003)</p> <p>Updated section B.1.4 "B.1.4 '<\$paratext[AppSectionSub]>'>" by adding the words "Newly Created" to reflect that ..//cisco.comp/rc-tools/ is a new directory. (June 2003)</p> <p>Added the "12.2T Header File Location Changes for CDE" section. (February 2003)</p>
Appendix C, "CPU Profiling"	Added "For more information on performance analysis, please see: http://wwwin-ses.cisco.com/performance/index.htm " (May 2003)
Appendix E, "Branch Integration & Sync Processes"	Added a new appendix on Cisco IOS branch integration and sync processes. (September 2003)

Changes in the Eleventh Edition (January 2003)

The following changes have been made in the eleventh edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.2 of the Cisco IOS software.

Chapter	Change
System Initialization	Added a note on the order of NVGENed commands in the "Cisco IOS Initialization Process" section. (November 2002)
Basic IOS Kernel Services	Updated the information on "process_safe" code in the "Disabling Blocking and Using Safe Blocking with process_safe_()" Functions" section. (December 2002)
Memory Management	Updated the CHUNK_FLAGS_* information in the "Chunk Manager" section. (September 2002)
Interfaces and Drivers	Added new information concerning dynamically-allocated Boolean/bit flags for common IOS data structures, the "Use Subblock Reference Counters", the "IDB Data Structure Shrinking" section, and "Subblocks and Private Lists" section. (October 2002)
Platform-Specific Support	Updated and reorganized chapter. (December 2002)
Interprocessor Communications (IPC) Services	Added the "Congestion Status Notification Capability" section. (November 2002)
ANSI C Library	Added information on ANSI C Programming Language documentation. (November 2002)
Timer Services	Added "Timer Wheel Timers" documentation. (September 2002)
Writing Cisco IOS Error Messages	Added advisory for the "Format" parameter under 19.3.2.3 "Defining Error Messages" that content is ignored after \n in printf strings by some scripting applications. (October 2002)
Debugging and Error Logging	Added the "Traceback Recording" section. (October 2002) Updated Event Trace information on storing and displaying trace buffer entries merged and sorted by timestamp in "What is the Event Tracer?", "How to Use the Event Tracer", and "Event Tracer CLI Commands". (September 2002)
MIB Infrastructure	Changed the title from "IF-MIB" to "MIB Infrastructure"; added "SNMP Notification Infrastructure" and "HA MIB Sync Infrastructure" sections; updated "Supporting Main Interfaces, Controllers, and Subinterfaces" and "Adding Support to Register or Deregister with IF-MIB" subsections; and added sample implementation documentation. (January 2003)
Security	Added the new "Security" chapter and the "AutoSecure" section. (December 2002)
Appendix C, "CPU Profiling"	Updated location of the profile postprocessing program in Appendix C, "Process the Profiler Output". (August 2002).

Changes in the Tenth Edition (July 2002)

The following changes have been made in the tenth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.2 of the Cisco IOS software.

Chapter	Change
Overview	Added "Cisco IOS Software" description. (July 2002)

Basic IOS Kernel Services	Replaced “Scheduler” chapter with “Basic IOS Kernel Services” chapter. (June 2002) Added comments to “Register a Process for Notification on a Timer” section that multiple processes cannot watch the same managed timer and why. (March 2002)
File System	Added “NVRAM Multiple File System” documentation. (July 2002).
CNS	Removed information on <code>cns_base64_to_binary()</code> , <code>cns_binary_to_base64()</code> , <code>cns_ea_destroy()</code> , <code>cns_ea_wait_for_wb()</code> , <code>create_dom()</code> , and <code>create_xml_parser()</code> functions. (May 2002)
Subsystems	Revised “Requirements” property information in “Subsystem Properties” section, removed obsolete <code>req</code> property and related information, added the EHSA subsystem class, and added information on defining subsystem requirements in makefiles in “Define a Subsystem” section. (April 2002)
Registries and Services	Revised organization of the chapter. Added a list of terms, the “Sample Registry Usage” section and the “Steps to Create and Use a Registry” section. (May 2002)
Debugging and Error Logging	Added The Receive Latency Trace Facility documentation. (April 2002)
Binary Trees	Added section “Set Up an RB Tree With a Key That Is Non-32 Bits”. (June 2002)
Command-Line Parser	Added “Adding Parser Return Codes” documentation. (June 2002)
Appendix A, “Writing Cisco IOS Code: Style Issues”	Added a note about unacceptable “white space only” changes in the “Presentation of the Cisco IOS Source Code” section and added information about discouraged assignments in conditional statements. (May 2002) Replaced “Specifying Default Cases” with “Switch Statements and Default Cases”. (June 2002)

Changes in the Ninth Edition (February 2002)

The following changes have been made in the ninth edition of the *Cisco IOS Programmer’s Guide/Architecture Reference*. This edition corresponds to releases up to and including Release 12.2 of the Cisco IOS software.

Chapter	Change
System Initialization	Added caution regarding subsystem sequencing to manipulate the order of NVGEN commands to section “Cisco IOS Initialization Process.” (December 2001)
Strings and Character Output	“Print a String to Nonvolatile Storage” section added. (October 2001)
IF-MIB	New section “Entity MIBs” added. (October 2001)
Scheduler	Fixed cross-references to tables. (November 2001) Added new section “ <code>Process_get_analyze()</code> and <code>process_set_analyze()</code> Guidelines” (November 2001) Added note to “Send a Message to a Process.” (November 2001) Changed 2nd argument in <code>process_create()</code> in the “Create a Process” section. (December 2001) Removed “Specify only one Process Awoken per Enqueue” section, “Order Lock Acquisition Based on Time Sequence Over Process Priority” section, “Specify only one Process Awoken per Boolean Set” section and “Specify Only One Process Awoken per Bit Field Set” section. (January 2002)
Memory Management	Added “Establish a Subalias Region” section. (October 2001) Expanded “Return Memory.” (November 2001) Added centralized malloc processing for malloc failures (November 2001) Added note to “Allocate and Return a Memory Chunk Element” section. (February 2002)
Pools, Buffers, and Particles	Modified section on “Pool Structure.” (November 2001) Updated Figure 5-1 and Figure 5-2 to reflect changes in pool structure. (November 2001)

Writing Cisco IOS Code: Style Issues	Updated Standard Indentation section in “A.4.1 ’<\$paratext[AppSectionsub]>’.” (October 2001)
Queues and Lists	Changed LIST_FLAG_INTERRUPT_SAFE and LIST_FLAG_AUTOMATIC constants to LIST_FLAGS_INTERRUPT_SAFE and LIST_FLAGS_AUTOMATIC in the “Modify an Existing List” section. (December 2001) Added new section “Indexed Object Lists” (December 2001). Corrected statement that the <code>data_insertlist()</code> function provides interrupt protection in the section “Add an Item to a Queue.” (January 2002)
Interfaces and Drivers	Changed interface.h to subblock.h. (December 2001)
Binary Trees	Updated “AVL Trees” section, and the “Manipulate Wrapped AVL Trees” section. (December 2001) Added “Manipulate AVLDup Trees” section. (January 2002)
Registries and Services	Expanded “Manipulate RETVAL Services” section. (January 2002)
Interprocessor Communications (IPC) Services	Added “Useful Commands for Debugging an IPC Component” component. (January 2002)
CNS	New chapter. (February 2002)

Changes in the Eighth Edition (September 2001)

The following changes have been made in the eighth edition of the *Cisco IOS Programmer’s Guide/Architecture Reference*. This edition corresponds to Release 12.2 of the Cisco IOS software.

Chapter	Change
Strings and Character Output	New section on Format IPv6 Addresses was added. (August 2001) Automore functions have been added to this chapter which had been inadvertently omitted. (July 2001)
IF-MIB	New section “MIB Persistence Infrastructure” added. (July 2001)
Switching	New section documenting FIB subblocks. Also a new multipart definition of “card.” (September 2001)
Scheduler	Revised chapter. (September 2001)
Memory Management	Added Managed Chunks (New in Release 12.2). (September 2001)
Pools, Buffers, and Particles	Revised section 5.9.7 “Copy a Particle-based Packet into a Contiguous Packet Buffer.” (September 2001)

Changes in the Seventh Edition (June 2001)

The following changes have been made in the seventh edition of the *Cisco IOS Programmer’s Guide/Architecture Reference*. This edition corresponds to Release 12.2 of the Cisco IOS software.

Chapter	Change
Scheduler	Corrected entire subsection:(3.4.11) “Delay a Process during Initialization”. Changed title for sections:(3.4.11) “Delay a Process during Initialization” and (3.1.5) “FYI:Dialer Backup/Dial-on-demand Routing Interfaces” and (3.1.4.3) “Enabling or Disabling Interface Keepalives.” (July 2001)

Chapter	Change
Switching	Updated section 23.3, "Adding a CEF Feature." (June 2001) In Table 23-1, "Switching Options on High-End Routers," changed "N" to "Y" for Distributed Optimum Switching on the Cisco 7000 w/RSP platform. (July 2000)
Writing Cisco IOS Error Messages	New chapter. Had been a technical note. The technical note was updated and then incorporated into this manual. (September 2000)
Debugging and Error Logging	Major new section: "The Event Trace Facility," describing a new subsystem added. (June 2001) Added new section: "What is the Enhanced Error Message Log Count". (July 2001) Added new sub-section: "Differences between buginf() and printf()" (July 2001) Added "Links to Other Debugging Documentation" section at the end of the chapter. (September 2000)
Registries and Services	Updated instructions in section: "Placement of xxx_registry.o in Makefiles." Added new section: "Service Types Usage Guidelines," provided to help you understand some of the limitations of service type usage, particularly STUB services. Updated the note in the section: "Manipulate STUB Services." (June 2001)
Command-Line Parser	Added two new sections: "" and "Debugging Parser Ambiguity." (June 2001)
High Availability (HA)	New chapter. (June 2001)
Binary Trees	Corrected statement describing insertions, deletions, and searches of "Red-Black (RB) Trees." (June 2001)
Porting Cisco IOS Software to a New Platform	Revisions throughout the chapter. (June 2001)
Writing Cisco IOS Code: Style Issues	Added new appendix section called Coding for Scalability. (July 2001)

Changes in the Sixth Edition (May 2000)

The following changes have been made in the sixth edition of the *Cisco IOS Programmer's Guide/Architecture Reference*. These changes do not correspond directly with those in Release 12.1 of the Cisco IOS software, which is mainly a bug fix release.s

Chapter	Change
Switching	Added two new sections: "Switching, an Overview," and "Adding a CEF Feature." (May 2000)
Small Feature Commit Procedure	New chapter. (May 2000)
Scheduler	Changed text from "TRUE" to "from FALSE to TRUE" in two places in the "Manage Booleans" section: "Boolean: Definition" and "Set the Value of a Watched Boolean." (May 2000) Added a note at the beginning of the chapter about two new functions and a new event, WATCHED_MESSAGE_EVENT. (May 2000)
Registries and Services	Added a new section, "Manipulate ILIST Services," and made sundry updates throughout the chapter, marked with change bars. (May 2000)
Writing, Testing, and Publishing MIBs	Updated the example code in "Establish a New MIB," in Step 6: "Update the sys/makemibs file to add the dependency for the new MIB." (October 1999)

Chapter	Change
Command-Line Parser	Changed text in “Negating Commands — the “No” and “Default” Keywords” to clarify that, in general, you must add a command explicitly to the config tree in order to enable the no prefix. (September 1999)
PDF	Added a PDF version of the <i>Cisco IOS Programmer’s Guide/Architecture Reference</i> . (http://wwwin-enged.cisco.com/ios/) (January 2000)
Memory Management	Added a note that <code>malloc()</code> functions return zero-filled memory. (May 2000)
Timer Services	Added new section, “Watching a Single Passive Timer.” (May 2000)

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:Dialer Backup/Dial-on-demand Routing Interfaces.” (June 1998)
Writing, Testing, and Publishing MIBs	Appended Cisco IOS Technical Note #2, <i>Testing and Publishing a MIB</i> , 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, “ <i>Using IDB Subblocks</i> ” in the section “IDB Terminology” (August 1998). Incorporated Cisco IOS Technical Note #7, “ <i>Subblock and VFT Infrastructure Changes</i> ” in the section “Subblock and VFT Support in 12.0” (January 1999). FYI: See the new chapter, “Extensible Plugin Driver API” in <i>Cisco IOS Device Drivers: Fundamentals of Architecture and Code</i> . (August 1998)
Memory Management	New section, “Virtual Memory.” (September 1998) Additions to table describing the <code>malloc()</code> Family of Functions. (November 1998)
System Initialization	New section: “Enhanced High System Availability (EWSA).” (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 “Negating Commands — the “No” and “Default” Keywords,” 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)

Chapter	Change
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's Guide/Architecture Reference*. These changes correspond to Cisco IOS Release 11.3.

Chapter	Change
Socket Interface	<p>New chapter. (November 1996)</p> <p>Revised the table of functions for new and revised functions, and to add macros. For details on changes to APIs, see the <i>preface to the Cisco IOS API Reference</i>. (November 1997)</p>
Timer Services	<p>A paragraph was added at the end of the section "Operation of Managed Timers." (November 1996)</p> <p>A paragraph was added to the end of the section "Initialize Managed Timers." (November 1996)</p> <p>In the section "Example: Managed Timers," the code in the example was changed. (November 1996)</p> <p>A paragraph was added to the sections "Guidelines for Allocating Memory" on page 20 and "Stop a Managed Timer" on page 15. (November 1997)</p>
Strings and Character Output	<p>The %CC format descriptor was added to the section "Format Time Strings." (November 1996)</p> <p>In Table 17-5, 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)</p>
Writing, Testing, and Publishing MIBs	<p>Updated RFCs to the current RFC numbers. (January 1997)</p> <p>In the "Textual Conventions" section, added new textual conventions. (January 1997)</p> <p>Rewrote the "Establish a New MIB" section, changing the procedure from CVS to ClearCase. (January 1997)</p> <p>Rewrote the "Compile a MIB" section. This section includes documentation for the new MIB compiler. (January 1997)</p>
Writing Drivers That Interface with the Cisco IOS Software	Removed this obsolete chapter. See instead the new manual on the subject: <i>Cisco IOS Device Drivers: Fundamentals of Architecture and Code</i> .
Interfaces and Drivers	<p>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)</p> <p>In the section "Scalability Changes," documented the change of MAX_INTERFACES from a global to a per-platform value. Provided example of the new way to define a structure and allocate space. (December 1997)</p>

Chapter	Change
Interprocessor 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 <code>xxx_registry.o</code> in makefiles, Added a section on the show registry support. (December 1997)
IF-MIB	New chapter. (December 1997)
Older Version of the Scheduler	Expanded information about the obsolete function <code>cfork()</code> 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 “MIB Infrastructure.” (December 1997)
Title	Added the phrase “Architecture Reference” to the title: <i>Cisco IOS Programmer’s Guide/Architecture Reference</i> . (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 <code>process_set_boolean()</code> function: <code>void process_set_boolean(watched_boolean *wb, boolean value);</code> 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 is visible to all devices on the PCI buses in a platform. It is an optional class.
	In the section “List of Common 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.

Chapter	Change
	<p>The section “Lock and Return Memory” has been added:</p> <p>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 <code>refcount</code>) of 1. To increment the <code>refcount</code> for a block of memory, use the mem_lock() function:</p> <pre>void mem_lock(void *memory);</pre> <p>To attempt to return a block of memory, use the <code>free()</code> function. All allocated memory can be returned using free().</p> <pre>void free(void *memory);</pre> <p>If <code>free()</code> is called with a block that has a <code>refcount</code> of 1, the block is returned to the memory pool from which it was created. If the <code>refcount</code> is greater than 1, <code>free()</code> decrements <code>refcount</code> 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, <code>free()</code> is the logical equivalent of <code>mem_unlock()</code> when using locked blocks of memory.</p>
	<p>The section “Example: Lock Memory” has been added.</p>
	<p>The default threshold for the <code>MEMPOOL_CLASS_LOCAL</code> pool class in the section “Set the Low- Memory Threshold” was incorrect. The section was corrected to the following:</p> <p>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 <code>MEMPOOL_CLASS_LOCAL</code> 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:</p> <pre>void mempool_set_fragment_threshold(mempool_class class, ulong size);</pre>
	<p>The default threshold for the <code>MEMPOOL_CLASS_LOCAL</code> pool class in the section “Set the Fragment Threshold” was incorrect. The section was corrected to the following:</p> <p>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 <code>MEMPOOL_CLASS_LOCAL</code> 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:</p> <pre>void mempool_set_low_threshold(mempool_class class, ulong size);</pre>
	<p>In the section “Create a Memory Chunk,” a description of the chunk pool flags was added.</p>

Chapter	Change
	<p>The section “Lock a Memory Chunk” was added: If the chunk pool was created with the <code>CHUNK_FLAGS_LOCKABLE</code> flag set, every element in the chunk pool has a reference count associated with it that can be incremented by the <code>chunk_lock()</code> function:</p> <pre>boolean chunk_lock(chunk_type *chunk, void*element);</pre> <p>The locking of chunk elements with <code>chunk_lock()</code> is exactly analogous to the <code>mem_lock()</code> function for data blocks allocated via <code>malloc()</code>, and the same examples and warnings apply.</p>
Platform-Specific Support	<p>In the section “Platform-Specific Strings,” the strings <code>PLATFORM_STRING_HARDWARE_REWORK</code> and <code>PLATFORM_STRING_LAST_RESET</code> have been added.</p>
	<p>In the section “Platform-Specific Values,” the value <code>PLATFORM_VALUE_LOG_BUFFER_SIZE</code> has been added.</p>
ANSI C Library	<p>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.</p>
Subsystems	<p>The subsystem class, <code>SUBSYS_CLASS_REGISTRY</code>, has been added.</p>
Registries and Services	<p>Major portions of the registries sections have been rewritten. The <code>registry_create()</code> function has been added.</p>
Timer Services	<p>The text in the section “Stop a Managed Timer” has been modified to the following: To stop a managed timer, use the <code>mgd_timer_stop()</code> 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.</p> <pre>void mgd_timer_stop(mgd_timer *timer);</pre> <p>A stopped timer is completely unlinked from the managed timer tree, so it is safe to free the memory containing the timer.</p>
Debugging and Error Logging	<p>The section “Example: Trace Buffer Leaks” was added.</p>
Binary Trees	<p>In the section “Initialize a Wrapped AVL Tree,” the syntax of the <code>wavl_init()</code> function was corrected to the following: To initialize a WAVL tree, use the <code>wavl_init()</code> function. In this function, you pass the <code>wavl_handle</code> 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.</p> <pre>boolean wavl_init(wavl_handle *handle, void *(*findblock)(wavl_node_type *), ...);</pre>
Writing Cisco IOS Code: Style Issues	<p>New chapter.</p>
CPU Profiling	<p>New chapter.</p>

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 Particle-based Buffer Management and Particle-based Buffer Pools: Overview was added. Some functions were renamed.
Interfaces and Drivers	Added inline functions in the section "Subblocks and Private Lists." Added the section "IDB Helper Functions."
Interprocessor 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 <code>print()</code> 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.

Table of Contents

Change History

- Changes in the Twenty-first Edition (*Month 200x*) v
- Changes in the Twentieth Edition (*March 2009*) viii
- Changes in the Nineteenth Edition (*December 2008*) ix
- Changes in the Eighteenth Edition (December 2007) x
- Changes in the Seventeenth Edition (April 2007) xii
- Changes in the Sixteenth Edition (July 2006) xv
- Changes in the Fifteenth Edition (September 2005) xvii
- Changes in the Fourteenth Edition (February 2005) xix
- Changes in the Thirteenth Edition (July 2004) xxii
- Changes in the Twelfth Edition (September 2003) xxvi
- Changes in the Eleventh Edition (January 2003) xxxi
- Changes in the Tenth Edition (July 2002) xxxi
- Changes in the Ninth Edition (February 2002) xxxii
- Changes in the Eighth Edition (September 2001) xxxiii
- Changes in the Seventh Edition (June 2001) xxxiii
- Changes in the Sixth Edition (May 2000) xxxiv
- Changes in the Fifth Edition (February 1999) xxxv
- Changes in the Fourth Edition (December 1997) xxxvi
- Changes in the Third Edition (September 1996) xxxvii
- Changes in the Second Edition (February 1996) xl

Figures xcvi

Tables ciii

About This Manual cix

- Document Objectives cix
- Audience cix
- Document Organization cix
- Document Conventions cxii

PART 1 Overview

Chapter 1 Overview 1-1

1.1	Cisco IOS Software	1-1
1.2	Network Service and Protocols	1-3
1.3	Cisco IOS Software Components	1-4
1.3.1	Kernel Services	1-5
1.3.1.1	Basic IOS Kernel Services	1-5
1.3.1.2	Memory Management	1-5
1.3.1.3	Pools, Buffers, and Particles	1-5
1.3.1.4	Interfaces and Drivers	1-6
1.3.1.5	Platform-Specific Support	1-6
1.3.1.6	Socket Interface	1-6
1.3.1.7	Interprocessor Communications (IPC) Services	1-7
1.3.1.8	File System	1-7
1.3.1.9	Standard Libraries	1-7
1.3.1.10	CNS	1-7
1.3.2	Kernel Support Services	1-7
1.3.2.1	Subsystems	1-7
1.3.2.2	Registries and Services	1-8
1.3.2.3	Timer Services and Time-of-Day Services	1-8
1.3.2.4	Strings and Character Output	1-9
1.3.2.5	Exception Handling	1-9
1.3.2.6	Debugging and Error Logging	1-9
1.3.3	Network Services	1-9
1.3.3.1	Binary Trees	1-9
1.3.3.2	Queues and Lists	1-9
1.3.3.3	Switching	1-10
1.3.3.4	High Availability (HA)	1-10
1.3.3.5	IP Services	1-10
1.3.4	Hardware-Specific Design	1-10
1.3.4.1	Porting Cisco IOS Software to a New Platform	1-10
1.3.5	Management Services	1-11
1.3.5.1	Command-Line Parser	1-11
1.3.5.2	Writing, Testing, and Publishing MIBs	1-11
1.3.5.3	MIB Infrastructure	1-11
1.3.5.4	Security	1-12
1.3.5.5	AAA	1-12
1.3.6	Other Useful Information	1-12
1.3.6.1	Scalable Process Implementation	1-12
1.3.6.2	Backup System	1-12
1.3.6.3	Verifying Cisco IOS Modular Images	1-12
1.3.6.4	Writing DDTs Release-note Enclosures	1-12
1.3.6.5	Small Feature Commit Procedure	1-13
1.3.6.6	Current Cisco IOS Initiatives	1-13
1.3.7	Appendices	1-13
1.3.7.1	Writing Cisco IOS Code: Style Issues	1-13
1.3.7.2	Cisco IOS Software Organization	1-13
1.3.7.3	CPU Profiling	1-14
1.3.7.4	Older Version of the Scheduler	1-14
1.3.7.5	Branch Integration & Sync Processes	1-14
1.3.7.6	Doxygen Instructions	1-14

1.4	Scalability Changes	1-14
1.4.1	Subblock and Lists	1-14
1.4.2	Extensible Plugin Driver API	1-15
1.4.3	Event-Driven Scheduling	1-15
1.4.4	IDB Data Structure Shrinking	1-15
1.4.4.1	Dynamically Allocated Boolean Flags for IDBs	1-15
1.4.5	IDB Subblock Modularity	1-15
1.4.6	Other Scalability Changes	1-16
1.5	Determining Infrastructure Changes in Releases	1-16
1.5.1	Major Infrastructure Changes	1-16
1.5.2	Library Changes	1-16
1.5.3	Feature Changes	1-16
1.5.4	Sample Infrastructure Changes on Newer Releases	1-17
1.5.5	Sample Infrastructure Changes on Older Releases	1-17
1.5.6	The CHANGES File	1-17
1.5.7	Related Infrastructure Initiatives	1-18
1.6	Searching Email Alias News Archives	1-18
1.7	Migrating to ION	1-19

Chapter 2 System Initialization 2-1

2.1	Overview: System Initialization	2-1
2.2	Basic Initialization	2-2
2.2.1	Initialization by the ROM Monitor	2-2
2.2.2	Bootstrap a Cisco IOS Image	2-2
2.2.2.1	Bootstrap a Cisco IOS Image from ROM	2-3
2.2.2.2	Bootstrap a Cisco IOS Image from a Network	2-3
2.2.2.2.1	Out of Sequence Response Code	2-4
2.2.2.3	Bootstrap a Cisco IOS Image from Flash Memory	2-4
2.2.2.4	Detailed Example of how the ROMMON Boots an Image	2-4
2.2.3	Allow the Cisco IOS Image to Take Control of the Platform	2-17
2.2.4	Fundamental Initialization	2-18
2.3	Cisco IOS Initialization Process	2-19
2.3.1	Parsing and Command Placement	2-21
2.4	Setting the Interrupt Levels	2-21
2.4.1	Cisco IOS Interrupt Levels	2-22
2.4.2	Getting the Current Interrupt Level	2-22
2.4.3	Raising the Current Interrupt Level	2-23
2.4.4	Resetting the Current Interrupt Level	2-24
2.4.5	Setting the Current Interrupt Level (IOS only)	2-24
2.5	Checking for Interrupt Status	2-25
2.6	Enhanced High System Availability (EHSA)	2-25
2.7	EHSA Overview	2-25
2.7.1	Master-Slave Communications	2-26
2.7.2	Health Monitoring	2-26
2.7.3	Slave Access and Information Requirements	2-26
2.7.3.1	File System	2-26
2.7.3.2	Boot Parameters	2-26
2.7.3.3	Time	2-27
2.7.3.4	Future Projects	2-27

2.7.3.5	Version Compatibility	2-27
2.7.3.6	Auto Sync	2-28
2.7.3.7	Slave Console	2-28
2.7.3.8	Slave Message Logging on Master	2-28
2.7.3.9	Seamless Software Upgrades	2-28
2.7.3.10	MAC Addresses	2-28
2.7.4	Basic Flow and Operation	2-29
2.7.4.1	Basic Slave Operation	2-29
2.7.4.2	Initialization	2-29
2.7.4.3	Interaction with the Boot Loader Image	2-29
2.8	EHSA Implementation Guide	2-30
2.8.1	Initializing EHSA	2-30
2.8.1.1	SUBSYS_CLASS_EHSA	2-30
2.8.2	EHSA APIs	2-30
2.8.2.1	Actions on Status-State Transitions	2-31
2.0.0.1	Using ehsa_event() to Trigger State Transitions	2-31
2.0.1	Examples	2-32
2.0.1.1	IPC Setup	2-33
2.0.1.2	Determining Primary/Secondary Status	2-33
2.0.1.3	Platform Initialization of EHSA Information and Vectors	2-33
2.0.2	The Secondary Background Process	2-34
2.0.3	The Primary Background Process	2-34
2.0.4	Changes in the Initialization Sequence	2-34
2.1	Common EHSA CLI	2-36
2.1.1	Platforms Currently Represented	2-36
2.1.2	General Redundancy (EHSA) CLI Syntax	2-37
2.1.2.1	Redundancy Configuration	2-37
2.1.2.2	Redundancy Display	2-37
2.1.2.3	Redundancy Operations	2-37
2.1.3	Santa (6400) Redundancy CLI	2-37
2.2	EHSA Crash Handling	2-38
2.2.1	Background	2-38
2.2.2	What Happens When a Primary Crashes?	2-38
2.2.3	What Happens When a Secondary Crashes?	2-39
2.2.4	Summary of Routines and Code Additions	2-39
2.3	IOS Warm Reboot	2-40
2.3.0.1	Main Benefits of IOS Warm Reboot	2-41
2.3.1	List of IOS Warm Reboot Terms	2-41
2.3.2	List of IOS Warm Reboot Functions	2-42
2.3.3	IOS Warm Reboot CLI	2-42
2.3.3.1	Warm-reboot config Command	2-43
2.3.3.2	Reload exec Command	2-43
2.3.3.3	Show warm-reboot Command	2-44
2.3.4	IOS Warm Reboot Flow Overview	2-44
2.3.4.1	Warm Reboot Platform-Independent Functions	2-45
2.3.4.2	Warm Reboot Platform-Independent Registry Services	2-45
2.3.4.3	Warm Reboot Platform-Specific Functions to Write	2-45
2.3.4.4	Processor-Specific Warm Reboot Support	2-48
2.3.5	IOS Warm Reboot Detailed System Flow	2-48
2.3.5.1	Enabling/Disabling IOS Warm Reboot	2-48
2.3.5.2	Warm-Reboot-Aware System Startup	2-49
2.3.5.2.1	Save the Startup Context	2-49

2.3.5.2.2	Save the Data Segment	2-49
2.3.5.2.3	Initialize the Platform Memory	2-49
2.3.5.3	Warm Reboot Exception Handling	2-50
2.3.5.3.1	Sample Standard Exception Handling	2-50
2.3.5.3.2	Changes in Exception Handling for Warm Reboot	2-50
2.3.5.3.3	Sample Exception Handling for Warm Reboot	2-51
2.3.5.3.4	Summary of Warm Reboot on an Exception	2-52
2.3.5.4	User-Initiated Warm Reboot	2-52
2.3.5.4.1	Summary of Warm Reboot through the CLI	2-53
2.3.6	Future IOS Warm Reboot Implementation	2-54
2.3.7	Data Structure Support for IOS Warm Reboot	2-54
2.3.8	How to Make a Platform IOS Warm-Reboot-Aware	2-54
2.3.8.1	IOS Warm Reboot Example: 7200	2-55
2.4	IOS Warm Upgrade	2-57
2.4.1	IOS Warm Upgrade Components	2-58
2.4.2	IOS Warm Upgrade API Functions	2-59
2.4.3	IOS Warm Upgrade CLI	2-59
2.4.3.1	Modifications to Existing CLIs	2-59
2.4.3.2	New CLI	2-59
2.4.4	IOS Warm Upgrade Flow Overview	2-59
2.4.4.1	Common Warm Upgrade Code	2-60
2.4.4.2	Processor-Specific Warm Upgrade Code	2-60
2.4.4.3	Platform-Specific Warm Upgrade Code	2-60
2.4.4.4	Image Format-Specific Warm Upgrade Code (only ELF is supported currently)	2-60
2.5	Retrieving System Information	2-61

PART 2 Kernel Services

Chapter 3 Basic IOS Kernel Services 3-1

3.1	Introduction	3-1
3.1.1	Terminology	3-2
3.1.2	List of Functions	3-4
3.2	Scheduler	3-6
3.2.1	Cisco IOS Task States	3-7
3.2.2	Scheduler and IOS Tasks	3-7
3.2.3	Cisco IOS Task Scheduling Characteristics	3-7
3.2.4	Scheduler Queues	3-7
3.2.4.1	Ready Queues	3-8
3.2.4.2	Idle Queues	3-8
3.2.4.3	Dead Queues	3-8
3.2.4.4	Moving Tasks between Queues	3-9
3.2.5	Scheduler Selection Algorithm	3-9
3.2.6	Important CLI Commands	3-15
3.3	Process Management	3-16
3.3.1	Cisco IOS Task Priorities	3-16
3.3.2	Cisco IOS Task States	3-17
3.3.3	Managing Cisco IOS Tasks	3-18
3.3.3.1	Overview	3-18
3.3.3.2	Creating a Cisco IOS Task	3-19
3.3.3.2.1	Examples	3-20
3.3.3.3	Starting a Cisco IOS Task	3-20

3.3.3.4	Suspending a Cisco IOS Task	3-20
3.3.3.5	Setting and Retrieving Information about a Cisco IOS Task	3-26
3.3.3.5.1	Process_get_analyze() and process_set_analyze() Guidelines	3-27
3.3.3.6	Determining Whether a Cisco IOS Task Exists	3-27
3.3.3.7	Destroying a Cisco IOS Task	3-27
3.3.4	Example	3-28
3.3.5	Process Accounting and CPU Utilization	3-29
3.3.5.1	Process Accounting Threshold Values	3-29
3.3.5.2	CPU Usage Accounting	3-29
3.3.5.3	Fixing CPU Usage Accounting Inaccuracies	3-29
3.3.6	How to Preempt IOS Processes	3-31
3.3.6.1	Flow of Preemptive Processes	3-31
3.3.6.1.1	Preemptive Processes and the Scheduler	3-33
3.3.6.1.2	Preemptive Processes and Timer Events	3-33
3.3.6.2	Changes to the sprocess Structure for the Pseudo-preemption Infrastructure	3-33
3.3.6.3	Pseudo-preemption Infrastructure Considerations and Restrictions	3-34
3.3.6.3.1	Pseudo-preemption Infrastructure Memory and Performance Considerations	3-34
3.3.6.3.2	Pseudo-preemption Infrastructure Configuration Considerations	3-35
3.3.6.3.3	Pseudo-preemption Infrastructure Restrictions	3-35
3.3.6.4	Preemptive Process Capabilities	3-35
3.3.6.5	API for the IOS Pseudo-preemption Infrastructure	3-36
3.3.6.5.1	Create a Preemptive Process	3-36
3.3.6.5.2	Disable Preemption	3-36
3.3.6.5.3	Enable Preemption	3-37
3.3.6.5.4	Create a Recursive Semaphore	3-37
3.4	Event Management	3-37
3.4.1	Managing Sets of Events	3-38
3.4.1.1	How Event Sets Are Used	3-38
3.4.2	Example	3-38
3.4.3	Managing Individual Events	3-39
3.4.3.1	Event Types	3-39
3.4.3.2	Event Operations	3-39
3.4.3.3	Queues: Definitions, Classes, and Attributes	3-41
3.4.3.3.1	Queue: Definition	3-41
3.4.3.3.2	Queue Operations	3-41
3.4.3.4	Managed Booleans	3-43
3.4.3.4.1	Managed Boolean: Definition	3-43
3.4.3.5	Managed Bit Fields	3-44
3.4.3.5.1	Managed Bit Fields: Definition	3-44
3.4.3.6	Managed Semaphores	3-45
3.4.3.6.1	Managed Semaphore: Definition	3-46
3.4.3.7	Atomic Locks	3-47
3.4.3.7.1	Locking and Unlocking an Atomic Lock: Example	3-47
3.4.3.8	Read/Write Locks	3-48
3.4.3.9	Managed Messages	3-48
3.4.3.10	Managed Timers	3-49
3.4.4	Event Management Topics	3-51
3.4.4.1	Directly Waking up a Cisco IOS Task	3-51
3.4.4.2	Determining Why a Task was Awakened	3-52
3.4.4.3	Waiting on Events	3-52
3.4.4.3.1	process_wait_for_event()	3-53
3.4.4.3.2	process_wait_for_event_timed()	3-55
3.4.4.4	Waiting Until System Initialization Completes	3-55

3.4.4.4.1	Wait Until System Initialization Completes: Example	3-55
3.4.4.5	Disabling Blocking and Using Safe Blocking with process_safe_() Functions	3-56
3.4.5	Useful Event Management Commands	3-58
3.5	Random Number Generation	3-59
3.5.1	Different Random Number Requirements	3-60
3.5.2	Non-Cryptographic pRNG Functions	3-60
3.5.2.1	Cisco IOS RNG Facility	3-61
3.5.3	IOS Universal Cryptographic pRNG Functions	3-61
3.5.4	Crypto Image-Only IOS pRNG Functions	3-61
3.5.5	Establishing an RNG Seed Context	3-62
3.5.5.1	Establishing an RNG Seed Context: Examples	3-62
3.5.6	Obtaining a Random Number	3-62
3.5.6.1	Obtaining a Random Number: Examples	3-63
3.5.7	Filling Memory with Random Numbers	3-64
3.5.7.1	Filling Memory with Random Numbers: Examples	3-64
3.5.8	Secure Hash Algorithm with get_random_bytes() Function	3-64
3.5.8.1	Secure Hash Algorithm with get_random_bytes() Function: Example	3-65
3.5.9	Further Information	3-65

Chapter 4 Memory Management 4-1

4.1	Overview: Memory Management	4-2
4.1.1	List of Memory Management Functions	4-2
4.1.2	Regions and the Region Manager	4-4
4.1.3	Memory Pools, Memory Pool Manager, and Free Lists	4-5
4.1.4	Chunk Manager	4-5
4.1.5	Relationship between Regions, Memory Pools, and Chunks	4-5
4.2	Regions	4-6
4.2.1	Regions: Definition	4-6
4.2.2	Region Classes: Definition	4-6
4.2.3	Region Hierarchies: Definition	4-7
4.2.4	Create a Region	4-8
4.2.4.1	Create a Region: Example	4-8
4.2.5	Set a Region's Class	4-8
4.2.5.1	List of Region Classes	4-8
4.2.6	Set Media Access Attributes	4-9
4.2.6.1	List of Media Access Attributes	4-9
4.2.6.2	Example: Media Access Attributes	4-10
4.2.7	Establish Region Hierarchy	4-10
4.2.7.1	Region Hierarchy Types	4-10
4.2.7.2	Region Hierarchy Example	4-11
4.2.8	Establish an Alias Region	4-11
4.2.8.1	Example: Establish an Alias Region	4-11
4.2.9	Establish a Subalias Region	4-11
4.2.10	Set Inheritance Attributes	4-12
4.2.10.1	List of Region Inheritance Flags	4-12
4.2.11	Search through Memory Regions	4-12
4.2.11.1	Example: Search through Memory Regions by Address	4-13
4.2.11.2	Example: Search through Memory Regions by All Attributes	4-13
4.2.12	Determine Whether a Region Class Exists	4-13
4.2.13	Determine a Region's Size	4-13
4.2.13.1	Example: Determine a Region's Size	4-14
4.2.14	Retrieve a Region's Attributes	4-14

4.3	Memory Pools	4-15
4.3.1	Overview: Memory Pools	4-15
4.3.2	Free Lists: Overview	4-15
4.3.3	Create a Memory Pool	4-16
4.3.3.1	Example: Create a Memory Pool	4-16
4.3.4	Add Regions to a Memory Pool	4-16
4.3.5	Set a Memory Pool's Class	4-17
4.3.5.1	Mandatory Memory Pool Classes	4-17
4.3.5.2	Aliasable Memory Pool Classes	4-17
4.3.5.3	List of Common Memory Pool Classes	4-17
4.3.6	Alias Memory Pools	4-17
4.3.6.1	Example: Alias Memory Pools	4-18
4.3.7	Create Alternate Memory Pools	4-18
4.3.7.1	Example: Create Alternate Memory Pools	4-18
4.3.8	Allocate Memory	4-18
4.3.8.1	Allocate Unaligned Memory	4-19
4.3.8.2	Allocate Aligned Memory	4-19
4.3.8.3	Comparison of Memory Allocation Functions	4-19
4.3.8.4	Guidelines for Allocating Memory	4-20
4.3.8.5	Displaying Memory Allocation Failures	4-22
4.3.8.6	Example: Allocate Memory	4-23
4.3.9	Return Memory	4-24
4.3.10	Lock and Return Memory	4-24
4.3.10.1	Example: Lock Memory	4-24
4.3.11	Add Free List Sizes	4-26
4.3.11.1	Example: Add Free List Sizes	4-26
4.3.12	Specify Low-Memory Actions	4-27
4.3.12.1	Set the Low- Memory Threshold	4-27
4.3.12.2	Set the Fragment Threshold	4-27
4.3.12.3	Determine Whether Memory Is Low	4-27
4.3.13	Search through Memory Pools	4-27
4.3.13.1	Example: Search through Memory Pools by Memory Pool Address	4-28
4.3.13.2	Example: Search through Memory Pools by Memory Pool Class	4-28
4.3.14	Retrieve Statistics about a Memory Pool	4-28
4.3.15	Memchecks Support	4-28
4.3.16	IOS Memory Scaling	4-28
4.4	Chunk Manager	4-29
4.4.1	Overview: Chunk Manager	4-29
4.4.2	Guidelines for Using the Chunk Manager	4-29
4.4.3	Create a Memory Chunk	4-30
4.4.3.1	Example: Create a Memory Chunk	4-33
4.4.3.2	Dynamic Chunk Sibling Creation and Destroy	4-33
4.4.3.2.1	A Description of Chunk Sibling Chaining	4-34
4.4.3.3	External Memory	4-34
4.4.4	Allocate and Return a Memory Chunk Element	4-34
4.4.4.1	Example: Allocate a Memory Chunk	4-36
4.4.5	Lock a Memory Chunk	4-36
4.4.6	Destroy a Memory Chunk	4-36
4.5	Managed Chunks (New in Release 12.2)	4-36
4.5.1	Setting Up a Managed Chunk	4-37
4.5.2	How Managed Chunks Work	4-37
4.5.3	Create a Managed Chunk	4-37

4.6	Transient Memory Allocation (New in 12.2S)	4-38
4.6.1	Dynamic Region Manager	4-38
4.6.2	Static and Transient Memory Pools	4-39
4.6.2.1	Managing Memory for Memory Pools and the Free-Region List	4-40
4.6.2.2	Freeing Memory to Add to freeregionlist	4-40
4.6.3	Transient Memory API Functions	4-40
4.6.3.1	Initializing the Free-Region List	4-40
4.6.3.2	Allocating Memory from the Transient Memory Pool	4-40
4.6.3.3	Allocating Aligned Memory from the Transient Memory Pool	4-41
4.6.4	Transient Chunks	4-41
4.6.5	Transient Memory and CLI Output	4-41
4.6.5.1	show region	4-41
4.6.5.2	show memory	4-42
4.7	Memory Management Commands	4-42
4.7.1	Determine Amount of Memory Available	4-45
4.7.1.1	Display Total System Memory	4-45
4.7.1.2	Display Memory for Each Region	4-46
4.7.2	Memory Display Commands	4-46
4.7.2.1	show memory Output	4-46
4.7.2.1.1	The blocktype Header and **previous	4-47
4.7.2.2	show memory free Output	4-48
4.7.2.3	show memory dead Output	4-48
4.7.2.4	show memory allocating-process Output	4-49
4.7.2.5	show memory debug references Output	4-50
4.7.2.6	show memory ecc Output	4-50
4.7.2.7	show memory fast Output	4-51
4.7.2.8	show memory fragment Output	4-51
4.7.2.9	show memory multibus Output	4-53
4.7.2.10	show memory pci Output	4-54
4.7.2.11	show memory processor Output	4-54
4.7.2.12	show memory scan Output	4-55
4.7.2.13	show memory transient Output	4-56
4.7.2.14	show processes memory Output	4-56
4.7.3	Memory Validation Commands	4-57
4.7.4	Malloc-Lite Memory Commands	4-58
4.7.5	How to Check Blocks and Buffers	4-58
4.7.6	Garbage Detector	4-59
4.7.6.1	Before Using GD	4-59
4.7.6.2	Using GD	4-59
4.7.6.3	GD Command-Line Interface	4-59
4.7.6.3.1	show memory debug leaks	4-59
4.7.6.3.2	show memory debug leaks summary	4-60
4.7.6.3.3	show memory debug leaks chunks	4-61
4.7.6.3.4	show memory debug leaks largest	4-63
4.7.6.3.5	show memory debug leaks lowmem	4-64
4.7.6.3.6	set memory debug incremental starting-time	4-64
4.7.6.3.7	show memory debug incremental status	4-64
4.7.6.3.8	show memory debug incremental leaks	4-64
4.7.6.3.9	show memory debug incremental leaks lowmem	4-64
4.7.6.3.10	show memory debug incremental allocation	4-64
4.7.6.4	False Alarms	4-65
4.8	Dynamic Bitfield Management	4-65
4.8.1	List of Bitlogic API Functions	4-66

4.8.2	Checking Bitfields	4-66
4.8.3	Clearing Bitfields	4-66
4.8.4	Destroying Bitfields	4-67
4.8.5	Setting Bitfields	4-67
4.8.6	Locking Bitfields	4-67
4.9	Dynamic Bitlists	4-68
4.9.1	The Bitlist API	4-68
4.9.2	Bitlist Wrapper Functions	4-69
4.9.3	Performance Analysis of Bitlist Algorithms	4-70
4.9.4	Safe Bitlist API Functions	4-71
4.10	Virtual Memory	4-71
4.10.1	Introduction to VM	4-72
4.10.1.1	The Paging Game: Rules	4-72
4.10.1.2	The Paging Game: Notes	4-73
4.10.2	Overview of Cisco IOS VM	4-73
4.10.2.1	Requirements	4-73
4.10.2.2	Benefits and Costs	4-73
4.10.3	Engineering Effort	4-74
4.10.4	VM Rules	4-75
4.10.5	VM Primer	4-75
4.10.5.1	Virtual Addresses vs. Physical Addresses	4-75
4.10.5.2	What is an "address interval"?	4-76
4.10.5.3	Advice on Using VM	4-76
4.10.6	Porting VM to a Platform	4-77
4.10.7	Wish List	4-78
4.10.8	Style Considerations	4-79
4.10.9	Basic VM Terms and Concepts	4-80
4.11	Caching Issues	4-83
4.11.1	Invalidated Cache	4-83
4.11.2	History of IOS Caching	4-85
4.11.3	Current IOS Caching Issues	4-85
4.12	ID Manager	4-86
4.12.1	ID Manager API	4-87
4.12.2	How to Use the ID Manager	4-88
4.12.2.1	How to Get a Sequence of IDs	4-88
4.12.2.2	How to Reserve a Specific ID in a Distributed System	4-89
4.12.3	LRU ID Manager	4-89
4.13	Shared Information Utility	4-89
4.13.1	Registration	4-91
4.13.1.1	Dataset Registration	4-91
4.13.1.2	Group Registration	4-91
4.13.2	Reading and Modes of Operation	4-92
4.13.2.1	Suspend Update Mode	4-92
4.13.2.2	Immediate Update Mode	4-93
4.13.2.3	Application Update Mode	4-93
4.13.3	Writing	4-93
4.13.3.1	Obtaining New Write View	4-93
4.13.3.2	Commit	4-94
4.13.3.3	Abort	4-94
4.13.3.4	Callbacks	4-94
4.13.3.5	Enabling and Disabling Commit	4-94

- 4.13.3.6 Multiple Writers 4-94
- 4.13.4 Notifications 4-95
- 4.13.5 Shared Information Utility API 4-96

Chapter 5 Pools, Buffers, and Particles 5-1

- 5.1 Buffer Management: Overview 5-1
 - 5.1.1 Terminology 5-2
 - 5.1.2 Packet Buffers 5-4
 - 5.1.3 Particle-based Buffers 5-5
- 5.2 Generic Pool Management 5-6
 - 5.2.1 Generic Pool Operation and Characteristics 5-6
 - 5.2.2 Pool Structure 5-6
 - 5.2.3 Pool Group and Item Size 5-7
 - 5.2.4 Pool Lists 5-7
 - 5.2.5 Static and Dynamic Pools 5-7
 - 5.2.6 Permanent and Temporary Items 5-8
 - 5.2.7 Create a Pool 5-8
 - 5.2.7.1 Pool Group Number and Creating Private Pools 5-9
 - 5.2.7.2 Pool Flags 5-10
 - 5.2.7.3 Pool List Structure 5-10
 - 5.2.7.4 Pool Item Function Vectors 5-11
 - 5.2.8 Populate or Adjust a Pool 5-12
 - 5.2.8.1 Pool Parameters in Dynamic Pools 5-13
 - 5.2.9 Obtain an Item from a Pool 5-14
 - 5.2.10 Return an Item to a Pool 5-14
 - 5.2.11 Pool Caches: Overview 5-14
 - 5.2.12 Structure of a Pool with a Cache 5-15
 - 5.2.13 Add a Pool Cache 5-15
 - 5.2.14 Fill a Pool Cache 5-16
 - 5.2.15 Obtain an Item from a Pool Cache 5-17
 - 5.2.16 Return an Item to a Pool Cache 5-17
 - 5.2.17 Destroy a Cache 5-17
 - 5.2.18 Handle Throttling Conditions in a Pool with a Cache 5-17
- 5.3 Packet Buffer Management 5-18
 - 5.3.1 Packet Buffer Headers 5-19
 - 5.3.2 Packet Buffer Data Blocks 5-19
 - 5.3.2.1 Organization Within a Data Block 5-19
 - 5.3.3 Packet Buffer Pools: Overview 5-21
 - 5.3.3.1 Public and Private Packet Buffer Pools 5-21
 - 5.3.4 Initialize System-wide Public Packet Buffer Pools 5-21
 - 5.3.5 Create a Public Packet Buffer Pool 5-22
 - 5.3.5.1 Example: Create a Public Packet Buffer Pool 5-23
 - 5.3.6 Create a Private Packet Buffer Pool 5-23
 - 5.3.6.1 Find and Assign a Fallback Packet Buffer Pool 5-24
 - 5.3.6.2 Example: Create a Private Buffer Pool 5-24
 - 5.3.7 Obtain a Packet Buffer 5-24
 - 5.3.7.1 Obtain a Packet Buffer from a Public Buffer Pool 5-25
 - 5.3.7.2 Obtain a Packet Buffer from a Private Buffer Pool 5-26
 - 5.3.8 Lock and Unlock a Packet Buffer 5-26
 - 5.3.9 Return a Packet Buffer to a Pool 5-26
 - 5.3.9.1 More Guidelines for Returning a Packet Buffer 5-27
 - 5.3.10 Reuse a Packet Buffer 5-28
 - 5.3.11 Prune a Packet Buffer Pool 5-28

5.3.12	Duplicate or Expand a Packet Buffer	5-28
5.3.12.1	Duplicate a Packet Buffer to a Given Destination Buffer	5-30
5.3.12.2	Duplicate and Recenter a Packet Buffer	5-31
5.3.12.3	Locate the Ideal Offset or Center Within a Packet Buffer	5-31
5.3.12.4	Comparison of Packet Buffer Duplication with and without Recentering	5-31
5.3.12.5	Duplicate a Packet Buffer into a New Destination Buffer	5-32
5.3.12.6	Allow the Size of a Packet Buffer to Grow	5-33
5.3.13	Packet Buffer Caches: Overview	5-33
5.3.14	Create and Populate a Packet Buffer Cache	5-33
5.3.14.1	Example: Create and Fill a Packet Buffer Cache	5-34
5.3.14.2	Example: Set up a Private Packet Buffer Pool with a Cache	5-34
5.3.15	Access Packet Buffers in a Cache	5-34
5.3.16	Manipulate Input Interface for Packet Processing	5-35
5.3.17	Associate a Packet Buffer with an Interface	5-37
5.3.18	Move a Packet Buffer to Another Interface	5-37
5.3.19	Remove a Packet Buffer from an Interface	5-38
5.3.20	Manipulate Packet Buffers on Indirect Queues	5-38
5.3.21	Packet Buffer Pools Code Example	5-38
5.4	Particle-based Buffer Management	5-39
5.4.1	Particles: Overview	5-39
5.4.2	Particle Structure	5-40
5.4.3	Particle-based Buffer Pools: Overview	5-41
5.4.4	Create and Fill a Particle Pool	5-45
5.4.5	Create and Fill a Particle Cache	5-45
5.4.6	Obtain a Particle from a Particle Pool	5-45
5.4.7	Manipulate the Reference Count of a Particle	5-46
5.4.8	Return a Particle to a Pool	5-46
5.4.9	Prune a Public Particle Pool	5-46
5.4.10	Add a Particle to a Particle-based Packet	5-46
5.4.11	Remove a Particle from a Particle-based Packet	5-47
5.4.12	Copy a Particle-based Packet into a Contiguous Packet Buffer	5-47
5.4.13	Packet Reparenting for Particles	5-48
5.4.14	Particle Clones	5-49
5.4.14.1	Specifying a Clone Queue When Cloning Particle-based Packets	5-51
5.5	Useful Buffer Commands	5-52
5.5.1	show buffers Output Example	5-54
5.5.2	show buffers Enabled Command Options	5-57

Chapter 6 Interfaces and Drivers 6-1

6.1	Introduction	6-1
6.1.1	Terminology	6-2
6.2	Manipulate IDBs	6-3
6.2.1	Terminology	6-3
6.2.2	Create an IDB	6-3
6.2.3	Link an IDB	6-5
6.2.4	Iterate over a List of IDBs	6-5
6.2.5	Delete an IDB	6-5
6.2.6	Reuse an IDB	6-6
6.2.7	ATM IDB Recycling	6-8
6.2.7.1	Removal of ATM IDB Data Structures	6-8
6.2.7.2	Implementation Details for Enabling Layer-2 ATM Subblock Reusability	6-9
6.2.8	IDB Protocol Counter API	6-9

6.2.8.1	IDB Protocol Counter Functions	6-9
6.2.8.2	IDB Protocol Counter Registry Functions	6-10
6.3	Subblocks and Private Lists	6-11
6.3.1	Subblock Identifier	6-12
6.3.2	Types of Subblocks	6-12
6.3.3	Which Type of Subblock to Use	6-13
6.3.3.1	Example: Creating a Subblock	6-13
6.3.3.2	Example: Retrieving a Subblock	6-13
6.3.3.3	Use Subblock Reference Counters	6-14
6.3.4	Common Subblock Header	6-15
6.3.4.1	Private IDB List	6-15
6.3.4.2	Subblock VFT	6-15
6.3.4.3	Cisco IOS Virtual Functions versus IDB Function Vectors	6-15
6.3.5	Implementation Details	6-16
6.3.6	Migration Path	6-18
6.3.6.1	Migration Example	6-18
6.3.6.2	Migrating Data from IDB to Subblock	6-18
6.3.6.3	Comparison of Subblocks and Private IDB Lists	6-18
6.3.7	Address Filter Function Vectors	6-19
6.3.7.1	Benefits of Using These Vectors	6-19
6.3.7.2	Address Filter Function Vectors Explained	6-19
6.3.7.3	Example Driver	6-20
6.4	Subblock Implementation Before Release 12.2S	6-20
6.4.1	Subblocks Types	6-21
6.4.2	Subblock Function Table	6-21
6.4.3	Add an IDB Subblock	6-21
6.4.4	Return a Pointer to an IDB Subblock	6-22
6.4.5	Traverse a List of Subblocks	6-22
6.4.6	Traverse Subblocks on an IDB	6-22
6.4.7	Release an IDB Subblock	6-22
6.4.8	Delete an IDB Subblock	6-22
6.5	Subblock Implementation After Release 12.2S	6-23
6.5.1	Subblock Identifier Changes	6-24
6.5.1.1	How Subblock Identifiers are Used	6-24
6.5.1.2	Modularity Solutions to Problems With Subblock Identifier Allocation	6-24
6.5.2	Subblock Modularity/Scalability Changes	6-25
6.5.3	Subblock Modularity Data Structure Changes	6-26
6.5.3.1	Subblock Function Table	6-26
6.5.3.2	Assign Subblock Identifiers Dynamically with an Indirect Identifier Field	6-28
6.5.3.3	Assign Subblocks in Fast Array or Subblock List	6-28
6.5.3.4	Subblock List Header Structure	6-29
6.5.4	API Changes	6-29
6.5.5	Add a New Subblock with Modularity Changes	6-30
6.5.6	Modularity Scope Issues	6-31
6.5.7	Guidelines to Convert to New Subblock Implementation	6-32
6.6	Manipulate a Private List of IDBs	6-33
6.6.1	Create a Private List of IDBs	6-34
6.6.2	Add an IDB to a Private List	6-34
6.6.3	Iterate a List of Private IDBs	6-34
6.6.4	Iterate a List of Private IDBs Safely	6-34
6.6.5	Remove an IDB from a Private List	6-35
6.6.6	Delete a Private List of IDBs	6-35

6.7	IDB Data Structure Shrinking	6-35
6.7.1	Dynamically Allocated Boolean Flags for IDBs	6-36
6.8	IDB Helper Functions	6-38
6.8.1	Apply a Function over a Private IDB List	6-38
6.8.2	Test an Interface for a Property	6-38
6.9	Encapsulate a Packet	6-38
6.10	Enqueue, Dequeue, and Transmit a Packet	6-39
6.11	Getting and Setting IDB Fields	6-39
6.11.1	Defining a Single Item list Parameter	6-40
6.11.2	Defining a Multiple Item list Parameter	6-40
6.11.3	Timestamps Within the list Parameter	6-41
6.11.3.1	Retrieving a Timestamp from the IDB	6-42
6.11.3.2	Saving a Timestamp into the IDB	6-42
6.11.4	Using the list Input Parameter	6-43
6.12	Protecting an IDB from the User	6-43
6.13	The linktype enum	6-44
6.14	nextsub Subinterface List (<i>new in 12.2T</i>)	6-46
6.15	Interface Locking Mechanism (<i>new in 12.1</i>)	6-47
6.15.1	Introduction	6-47
6.15.2	Design	6-47
6.15.3	Example of Interface Locking	6-47
6.15.4	Example of Using Interface Locking to Service Unblocked Interfaces	6-48
6.15.5	Detection of Interface Access without Lock	6-48
6.15.6	Data Structures	6-49
6.15.7	End User Interface	6-49
6.15.8	Adding Interface Locking Support on an Interface	6-49
6.15.9	Steps to Enable Interface Locking Support	6-49
6.16	The Dev-Object Model	6-50
6.16.1	Interacting with Dev Objects	6-50
6.16.2	Introduction to Using the Dev-Object Model	6-51
6.16.2.1	The Dev-Object Model Code Directory Structure	6-51
6.16.2.2	Rules for "C" #include Directives	6-52
6.16.2.3	Dev-Object Model API	6-52
6.17	Default Box-Wide IEEE MAC Address	6-56
Chapter 7	Platform-Specific Support	7-1
7.1	Platform-Specific Initialization: Overview	7-2
7.2	main() -> Fundamental Initialization	7-4
7.2.1	platform_main() Entry Conditions	7-4
7.2.2	platform_main() Tasks	7-4
7.2.3	Examples: Fundamental Initialization	7-6
7.2.3.1	ubr7200 platform_main()	7-6
7.2.3.2	AS5800 "Nitro" platform_main()	7-6
7.3	main() -> Memory Initialization	7-7
7.3.1	platform_memory_init() Entry Conditions	7-7
7.3.2	platform_memory_init() Tasks	7-7
7.3.3	Example: Memory Initialization	7-8
7.3.3.1	ubr900 platform_memory_init()	7-8

7.4	main() -> Exception Initialization	7-10
7.4.1	platform_exception_init() Entry Conditions	7-10
7.4.2	platform_exception_init() Tasks	7-10
7.4.3	Examples: Exception Initialization	7-12
7.4.3.1	c1000 platform_exception_init()	7-12
7.4.3.2	c7140 platform_exception_init()	7-12
7.5	main() -> Console Initialization	7-14
7.5.1	console_init() Entry Conditions	7-14
7.5.2	console_init() Tasks	7-15
7.5.3	Example: Console Initialization	7-15
7.5.3.1	c2600 console_init()	7-16
7.6	init_process() -> Packet Buffer Initialization	7-18
7.6.1	platform_buffer_init() Entry Conditions	7-18
7.6.2	platform_buffer_init() Tasks	7-18
7.6.3	Example: Packet Buffer Initialization	7-19
7.6.3.1	c6400 NRP Network Route Processor platform_buffer_init()	7-20
7.7	init_process() -> Non-volatile Storage Initialization	7-21
7.7.1	nv_init() Entry Conditions	7-21
7.7.2	nv_init() Tasks	7-21
7.7.3	Example: Non-volatile Storage Initialization	7-22
7.7.3.1	c10k nv_init()	7-22
7.7.4	platform_nvvar_support()	7-23
7.8	init_process() -> Interface Initialization	7-23
7.8.1	platform_interface_init() Entry Conditions	7-23
7.8.2	platform_interface_init() Tasks	7-23
7.8.3	Example: Interface Initialization	7-24
7.8.3.1	IAD2402 platform_interface_init()	7-25
7.8.4	init_process() -> Port Adapter Initialization	7-25
7.8.5	pas_slots_init() Entry Conditions	7-26
7.8.6	pas_slots_init() Tasks	7-26
7.8.6.1	IGX8400 pas_slots_init()	7-27
7.8.6.2	PA Function Calls	7-27
7.8.6.2.1	PA Functions Called from Interface Initialization	7-27
7.8.6.2.2	Platform-Specific Functions Called from Platform or PA Code	7-28
7.8.6.2.3	PA Functions Called from Platform or PA Code	7-28
7.8.7	init_process() - Memory Allocation	7-28
7.8.8	platform_pcimempool() Entry Conditions	7-28
7.8.9	platform_pcimempool() Tasks	7-29
7.8.9.1	Channelized T3 platform_pcimempool()	7-29
7.8.9.2	Defining Platform-Specific Memory Pools	7-30
7.8.9.3	Slot-Specific Selection of Memory Pools	7-30
7.9	init_process() -> Filesystem Initialization	7-30
7.9.1	platform_file_init() Entry Conditions	7-31
7.9.2	platform_file_init() Tasks	7-31
7.9.3	Example: Filesystem Initialization	7-31
7.9.3.1	AS5850 "Nitro" platform_file_init()	7-31
7.10	init_process() -> Terminal Line Initialization	7-32
7.10.1	platform_line_init() Entry Conditions	7-32
7.10.2	platform_line_init() Tasks	7-32
7.10.3	Example: Terminal Line Initialization	7-33
7.10.3.1	c1000 platform_line_init()	7-33

7.11	init_process() -> Configuration Verification Initialization	7-35
7.11.1	platform_verify_config() Entry Conditions	7-35
7.11.2	platform_verify_config() Tasks	7-35
7.11.3	Example: Configuration Verification Initialization	7-35
7.11.3.1	c7200 platform_verify_config()	7-36
7.12	init_process() Final Tasks	7-36
7.13	Platform-Specific Strings	7-37
7.13.1	Example: Calling Platform-Specific Strings	7-38
7.13.2	Example: Defining Platform-Specific Strings	7-39
7.14	Platform-Specific Values	7-40
7.14.1	Example: Calling Platform-Specific Values	7-41
7.14.2	Example: Defining Platform-Specific Values	7-41
7.15	Interface ID Format	7-41
7.15.1	Slotunit, Subunit, and Unit Values	7-43
7.15.2	Specifying Interface Identifier Format	7-43
7.15.3	One-level Standard or "Unit"	7-44
7.15.4	Two-level Standard or "Slots"	7-44
7.15.5	Three-level Standard with "shelf"	7-44
7.15.6	c7500 RSP Three-level Alternative with "slotunit"	7-45
7.15.6.1	The C7500 Route Switch Processor	7-45
7.15.6.2	The C7500 VIP	7-46
7.15.7	Catalyst 5K Three-level Alternative with "slotunit"	7-46
7.15.8	ONS155xx Optical Network System Alternative	7-46
7.15.9	Other Alternate Implementations	7-47
7.15.9.1	Other usages of hwidb->subunit	7-47
7.15.9.2	Documenting your platform identification implementation	7-47
7.15.10	Use other methods for IDB navigation	7-47
7.16	How to Develop New Platform Support	7-47
7.17	How to Request Platform-Specific Action from Platform-Independent Code	7-48
7.17.1	For a Platform-Specific Action Implemented on All Platforms	7-48
7.17.2	Implementing Optional Actions	7-49
7.17.2.1	For One-to-One or One-to-None Mapping	7-50
7.17.2.2	For One-to-Many or Many-to-One Mapping	7-50
7.17.2.2.1	One-to-Many Mapping	7-50
7.17.2.2.2	Many-to-One Mapping	7-51
7.17.3	For a Platform-Specific Action Implemented on One Platform	7-52

Chapter 8 Interprocessor Communications (IPC) Services 8-1

8.1	Introduction	8-1
8.2	IPC Services: Overview	8-2
8.3	Operational Environment	8-2
8.4	IPC Communication: Overview	8-3
8.5	IPC Terminology	8-4
8.6	Port Naming Services	8-5
8.6.1	Port Name Resolution	8-5
8.6.2	Port Name Syntax	8-5
8.6.2.1	Example: Port Name Syntax	8-6
8.6.2.2	Reserved Port Names	8-6

8.7	IPC Message Format	8-6
8.8	Unreliable IPC Messaging	8-9
8.9	Unreliable IPC Messaging with Notification	8-10
8.10	Reliable IPC Messaging	8-12
8.11	Remote Procedural Call (RPC) Messaging	8-16
8.12	IOS-IPC Process	8-18
8.13	IPC Multicast Messages	8-18
8.13.1	Multi-Operating System Support	8-18
8.13.2	Asynchronous Control Messaging	8-19
8.13.3	Functional Structure	8-19
8.13.4	System Flow	8-19
8.13.4.1	Control Messages	8-19
8.13.4.2	IPC-Platform Interaction	8-20
8.13.4.2.1	Initialization	8-20
8.13.4.2.2	Driver Operations	8-20
8.13.5	IPC Multicast Message Format	8-21
8.13.6	IPC Message Type Definition	8-21
8.14	Manipulate the Seat Table	8-22
8.14.1	Seat Table: Description	8-23
8.14.2	Create a Seat	8-23
8.14.3	Get Information about a Seat	8-23
8.14.4	Reset a Seat	8-24
8.15	Manipulate the Port Table	8-24
8.15.1	Port Table: Description	8-24
8.15.2	Create a Port	8-25
8.15.3	Register a Port	8-26
8.15.4	Open a Port	8-26
8.15.5	Find a Port	8-30
8.15.6	Close a Port	8-30
8.15.7	Remove a Port	8-31
8.16	Manipulate the Message Retransmission Table	8-32
8.16.1	Message Retransmission Table: Description	8-32
8.17	Send IPC Messages: Overview	8-32
8.17.1	Allocate a Message	8-33
8.17.2	Send a Message	8-33
8.17.3	Return a Message to the IPC System	8-35
8.18	Receive IPC Messages: Overview	8-35
8.18.1	Receive a Message	8-35
8.19	Simulate RPCs	8-38
8.20	Congestion Status Notification Capability	8-38
8.20.1	IPC Flow Control Signals	8-39
8.20.2	Sending Critical Messages During the STOP State	8-39
8.20.3	Congestion Status Notification Implementation	8-39
8.20.4	Policing the IPC Congestion State	8-40
8.20.4.1	Sending the STOP_SENDING_WARNING Signal	8-40
8.20.5	How are Nonsubscribing IPC Clients Handled?	8-41
8.20.6	How Long Can IPC Remain Congested?	8-41

8.21	Write an IPC Application	8-41
8.21.1	Create a Port	8-41
8.21.2	Open a Connection to the Port	8-42
8.21.3	Send a Message	8-42
8.21.3.1	Send a Message in Blocking Mode	8-42
8.21.3.2	Send a Message in Nonblocking Mode	8-43
8.22	Implementing IPCs on the RSP Platform	8-43
8.22.1	IPC CiscoBus Driver: Overview	8-43
8.22.2	IPC Setup Procedure	8-44
8.22.2.1	Discovery Phase	8-44
8.22.2.2	Initialization Phase	8-45
8.22.2.3	Registration Phase	8-48
8.22.3	Invoke the IPC Setup Procedure	8-49
8.22.4	Microcode Reload Handling	8-49
8.22.5	Implementation of the IPC CiscoBus Interface	8-49
8.22.5.1	Transmit Path	8-49
8.22.5.2	Receive Path	8-50
8.22.6	IPC Name Service	8-50
8.23	Useful Commands for Debugging an IPC Component	8-51
8.23.1	clear ipc stat	8-51
8.23.2	show ipc status	8-51
8.23.2.1	show ipc status cumulative	8-53
8.23.3	show ipc queue	8-53
8.23.4	show ipc ports	8-54
8.23.4.1	show ipc session	8-54
8.23.5	show ipc nodes	8-54
8.23.6	show ipc rpc	8-55
8.23.7	show ipc zones	8-55
8.23.8	show ipc session tx verbose	8-55
8.23.9	show ipc session rx verbose	8-57
8.24	IPC Master	8-58
8.24.1	Terms	8-58
8.24.2	What is the IPC Master?	8-58
8.24.3	IPC Master API Functions	8-59
8.24.3.1	Creating a Zone	8-59
8.24.3.2	Getting a Zone by Name	8-59
8.24.3.3	Getting Zone Data	8-60
8.24.3.4	Getting the Zone Name	8-60
8.24.3.5	Closing All Seats for a Zone	8-60
8.24.3.6	Removing a Zone	8-60
8.24.3.7	Setting a Zone's Platform ACK Vector	8-60
8.24.3.8	Setting a Zone's Platform Master Control Port ID	8-60
8.24.3.9	Setting a Zone's Platform Seat ID	8-60
8.24.3.10	Setting a Zone's Platform Seat Name	8-61
8.24.3.11	Setting a Zone's Platform Transport	8-61
8.24.3.12	Setting a Zone's Platform Transport Type	8-61
8.24.3.13	Setting a Zone's Platform TX Vector	8-61
8.24.3.14	Setting the Seat Master	8-61
8.24.3.15	Setting a Zone's Local Seat Address	8-61
8.25	IPC Get Port Notifications	8-61
8.25.1	Getting Notification of a Receiver Port Event	8-62
8.25.2	Getting Notification of a Sender Port Event	8-62

8.26 References 8-62

Chapter 9 File System 9-1

9.1 Overview 9-1

- 9.1.1 Application Level API 9-1
- 9.1.2 Classes of File Systems 9-2
- 9.1.3 File System Types 9-2
- 9.1.4 File System Features 9-3
- 9.1.5 File System Flags 9-3

9.2 Accessing File Systems 9-3

9.3 Implementing Simple File Systems 9-3

- 9.3.1 A Trivial IFS/File System 9-4
 - 9.3.1.1 Defining a File System 9-4
 - 9.3.1.2 Defining a File 9-5
 - 9.3.1.3 Example 1 - Reading a File 9-5
 - 9.3.1.4 Example 2 - A More Complex Read 9-7
 - 9.3.1.5 Example 3 - Writing a File 9-8
 - 9.3.1.6 Example 4 - Accessing the File System 9-11
 - 9.3.1.7 Example 5 - Reading a File 9-13
 - 9.3.1.8 Example 6 - Writing to a File 9-13
 - 9.3.1.9 Example 7 - Seeking a Position within a File 9-14
 - 9.3.1.10 Example 8 - Renaming a File 9-15
 - 9.3.1.11 Example 9 - Removing a File 9-16
 - 9.3.1.12 Example 10 - Making and Removing a Directory 9-17
 - 9.3.1.13 Example 11 - Collect File System Statistics 9-18
 - 9.3.1.14 Example 12 - IOCTL Function Support and Getting Next Directory Entry 9-19
 - 9.3.1.15 Example 13 - Closing All Files 9-20
- 9.3.2 Other Features 9-21
 - 9.3.2.1 Directories 9-21
 - 9.3.2.2 Timestamps 9-22

9.4 Implementing Complete File Systems 9-22

- 9.4.1 IFS/File System API 9-22
- 9.4.2 Common Data Structures 9-23
- 9.4.3 Implementation 9-24

9.5 Additional File System Hooks 9-26

- 9.5.1 Copy Prompt Hook 9-26
- 9.5.2 Copy Behavior Hook 9-27
- 9.5.3 show flash Hook 9-28

9.6 NVRAM API 9-28

- 9.6.1 Parser CLI Interface Functions 9-29
- 9.6.2 Platform-Specific Functions 9-30
- 9.6.3 NVRAM Functions Called by Platform-Specific Functions 9-31

9.7 Persistent Variable Method 9-31

- 9.7.1 Architecture 9-33
 - 9.7.1.1 Module Description 9-33
 - 9.7.1.1.1 Clients 9-33
 - 9.7.1.1.2 Persistent Variable Component 9-34
 - 9.7.1.1.3 Persistent Variable High Availability (HA) Subcomponent 9-35
 - 9.7.1.1.4 IFS Persistent media module 9-35
 - 9.7.1.1.5 NV IFS and NVRAM driver changes 9-35
 - 9.7.1.1.6 Makefile Changes 9-36

9.7.2	Restrictions	9-36
9.7.2.1	Software Restrictions and Considerations	9-36
9.7.2.2	Hardware Restrictions and Considerations	9-36
9.7.2.3	External Restrictions and Configuration	9-36
9.7.2.4	Initialization Configuration and Restrictions	9-37
9.7.3	Initialization	9-37
9.7.4	Internal Operations	9-37
9.7.4.1	Calling the API During Initialization	9-38
9.7.4.2	Calling the API Before Reset	9-38
9.7.4.3	Calling the API for Disabled Interrupts	9-39
9.7.4.4	Calling the API for Interrupt Context	9-39
9.7.5	Cache Flushes	9-39
9.7.6	Data Structures	9-39
9.7.6.1	Cache	9-40
9.7.6.2	Handle Context	9-40
9.7.7	Description of Algorithms	9-40
9.7.7.1	Cache Memory Allocation	9-41
9.7.7.1.1	Initialization	9-41
9.7.7.1.2	Insert	9-41
9.7.7.1.3	Delete	9-41
9.7.8	Internal Variables	9-41
9.7.9	Persistent Variable Media Module	9-41
9.7.10	HA Client	9-43
9.7.11	Interface Design	9-43
9.7.11.1	Makefile Changes	9-43
9.7.11.2	Persistent Variable Registry Interface	9-45
9.7.11.2.1	Virtual Switch Support	9-47
9.7.11.3	Media Registry Interface	9-47
9.7.12	Memory and Performance Impact	9-49
9.7.13	Source Code	9-50
9.8	References	9-51

Chapter 10 Socket Interface 10-1

10.1	SCTP Sockets API	10-3
10.1.1	New SCTP Sockets API Functions	10-3
10.1.2	Cisco-Specific SCTP Sockets Options	10-4
10.1.3	SCTP One-to-One Code Example	10-4
10.1.4	SCTP One-to-Many Code Example	10-8
10.2	Useful Commands	10-9

Chapter 11 Standard Libraries 11-1

11.1	Before the Security Initiative	11-1
11.2	Security Initiative	11-2
11.2.1	Mapping Between Unsafe Functions and Safe Functions	11-3
11.3	Safe Library Functions	11-7
11.3.1	Handling Run-time Constraints	11-8
11.3.2	Comparing Values in Memory	11-9
11.3.3	Copying a Region of Memory	11-9
11.3.4	Copying a Block of Memory, Handling the Overlap	11-9
11.3.5	Initializing an Area of Memory to a Desired Value	11-9
11.3.6	Zeroing Bytes	11-9
11.3.7	Comparing Strings by Converting Them to Uppercase	11-9

11.3.8	Locating the First Occurrence (Case-sensitive) of a Substring	11-9
11.3.9	Appending a Copy of a String	11-10
11.3.10	Comparing Two Strings	11-10
11.3.11	Comparing Two Character Arrays	11-10
11.3.12	Copying a String	11-10
11.3.13	Copying from One Character Array to Another	11-10
11.3.14	Copying Characters from a String to an Array	11-10
11.3.15	Copying Characters from an Array to a String	11-10
11.3.16	Counting the Number of Characters That Are Not in a String	11-11
11.3.17	Retrieving a Pointer to the First Occurrence of a Character in a String	11-11
11.3.18	Retrieving the Index of the First Character That Is Different Between Two Strings	11-11
11.3.19	Retrieving the Index of the First Character That Is the Same Between Two Strings	11-11
11.3.20	Checking Whether the Entire String Contains Alphanumeric Characters	11-11
11.3.21	Checking Whether the Entire String Contains ASCII	11-11
11.3.22	Checking Whether the Entire String Contains Digits	11-12
11.3.23	Checking Whether the Entire String Contains Hexadecimal Characters	11-12
11.3.24	Checking Whether the Entire String Is Lowercase	11-12
11.3.25	Checking Whether the Entire String Is Mixed Case	11-12
11.3.26	Validating the Makeup of a Password String	11-12
11.3.27	Checking Whether the Entire String Is Uppercase	11-12
11.3.28	Retrieving a Pointer to the Last Occurrence of a Character in a String	11-12
11.3.29	Retrieving the Index of the Last Character That Is Different Between Two Strings	11-13
11.3.30	Retrieving the Index of the Last Character That Is the Same Between Two Strings	11-13
11.3.31	Removing the Beginning White Space from a String by Shifting the Text Left	11-13
11.3.32	Concatenating Two Strings	11-13
11.3.33	Copying a Counted Nonoverlapping String	11-13
11.3.34	Computing the Length of a String	11-13
11.3.35	Retrieving a Pointer to the First Occurrence of Any Character Contained in One String That Is Also Contained in Another String	11-14
11.3.36	Determining Whether the Prefix Pointed to by One String Is at the Beginning of Another String	11-14
11.3.37	Removing Beginning and Trailing White Spaces From a String	11-14
11.3.38	Computing the Prefix Length of a String	11-14
11.3.39	Locating the First Occurrence of a Substring Regardless of Case	11-14
11.3.40	Retrieving the Next Token from a String	11-14
11.3.41	Scanning the String Converting Uppercase Characters to Lowercase Characters	11-15
11.3.42	Scanning the String Converting Lowercase Characters to Uppercase Characters	11-15
11.3.43	Nulling a String	11-15
11.4	Standard Library Support	11-15
11.4.1	ANSI C Library Functions	11-15
11.4.1.1	ANSI C Standard Header Files	11-15
11.4.2	POSIX Library Functions	11-16
11.4.2.1	POSIX Standard Header Files in Cisco IOS Source Code	11-32
11.4.2.2	Accessing POSIX Standards Reference Documentation	11-32
11.5	Absence of Floating Point Functions	11-35

Chapter 12 CNS 12-1

12.1	Introduction	12-1
12.1.1	Terms	12-1
12.1.2	CNS Overview	12-2
12.1.3	Event Agent Services	12-2
12.1.3.1	Starting an Event Agent Session	12-2
12.1.3.2	Sending an Event	12-3

12.1.3.2.1	Example	12-3
12.1.3.3	Receiving an Event	12-4
12.1.3.4	Waiting for Messages	12-4
12.1.3.5	Registering a Callback Function	12-4
12.1.3.6	Setting the Context	12-5
12.1.3.7	Storing the Callback Function Pointer	12-5
12.1.3.8	Clearing the Restart Callback Notification	12-5
12.1.3.9	Setting Up a Watched Boolean	12-5
12.1.3.10	Reading Received Event Subjects and Payloads	12-5
12.1.3.11	Stopping Receiving Events	12-6
12.1.3.12	Freeing Storage	12-6
12.1.3.13	Stopping an Event Agent Service Session	12-6
12.1.3.14	Notification When the Event Agent Shuts Down	12-6
12.1.4	Considerations for Using the Event Agent Service	12-6
12.1.4.1	Calling ea_free() for Each ea_read()	12-6
12.1.4.2	Unsubscribing to All Subjects No Longer Required	12-6
12.1.4.3	Calling ea_close() to Terminate the Event Agent Services Gracefully	12-7
12.1.4.4	Not Supporting Wild Card Subject Names	12-7
12.1.5	Config Agent Services	12-7
12.1.5.1	Registering a Callback Function Invoked by the Config Agent	12-7
12.1.5.2	Unregistering a Callback Function Invoked by the Config Agent	12-7
12.1.5.3	Commanding the Config Agent to Assign the config_id to the Value of the Argument String	12-7
12.1.6	Related Reading	12-7

PART 3 Kernel Support Services

Chapter 13 Subsystems 13-1

13.1	Overview: Subsystems	13-1
13.2	Runtime Subsystems	13-2
13.2.1	Defining a Runtime Subsystem	13-2
13.2.2	Subsystem Name	13-2
13.2.3	Subsystem Entry Point	13-2
13.2.4	Subsystem Initialization Sequencing	13-3
13.2.5	Subsystem Classes	13-3
13.2.5.1	How to Choose a Subsystem Class	13-5
13.2.6	Intra-Class Sequencing Property	13-5
13.2.6.1	Subsystem Sequencing Property Definitions: Example	13-5
13.3	Compile-time Subsystems	13-5
13.4	Subsystem Properties	13-5
13.4.1	Subsystem Property Definitions	13-6
13.4.2	Sequencing Property	13-6
13.4.2.1	Driver Initialization Sequence	13-7
13.4.3	Requirements Property	13-7
13.4.4	Error Messages	13-7
13.5	Define a Subsystem	13-8
13.5.1	Examples: Define a Subsystem	13-9
13.6	Fill In the Subsystem Structure	13-9
13.7	Tips for Creating a Subsystem	13-10
13.7.1	Create a New Subsystem	13-10

13.7.2	Rework System Processes	13-12
13.7.3	Reexamine Header File Dependencies	13-12
13.7.4	Use New IDB Subblocks to Store Private Variables	13-12

Chapter 14 Registries and Services 14-1

14.1	Introduction	14-1
14.1.1	Terms	14-2
14.1.2	Types of Services	14-2
14.1.3	Overview	14-4
14.1.3.1	registry, service, service point	14-4
14.1.4	Common Uses Of Registries	14-6
14.1.4.1	What Are the Two Common Uses of Registries?	14-6
14.1.4.2	Two Other Ways to Think of Registries	14-6
14.1.4.3	Words of Guidance When Adding New Registry Entries	14-6
14.1.5	Default Functions	14-7
14.1.6	REMOTE Registries in IOS	14-8
14.2	Sample Registry Usage	14-8
14.2.1	System Registries	14-9
14.2.2	Registry Usage Dependencies	14-12
14.3	Registry Files	14-13
14.4	The Registry Compiler	14-14
14.4.1	Registry Compilation Process	14-14
14.4.2	.reg File Metalanguage	14-14
14.4.2.1	Example: .reg File Format	14-15
14.4.3	.h File Contents	14-17
14.4.4	.c File Contents	14-18
14.5	Steps to Create and Use a Registry	14-19
14.5.1	Problems with Registry Files	14-21
14.5.1.1	Missing Definition	14-21
14.5.1.2	Incorrect Macro Usage	14-22
14.5.2	Placement of xxx_registry.o in Makefiles	14-22
14.6	Registry Services	14-23
14.6.1	Service Types Usage Guidelines	14-23
14.6.1.1	Limitations of STUB Service Type	14-23
14.6.1.2	Comparison of Service Type Functionality	14-24
14.6.2	show registry Support	14-24
14.7	Manipulate LIST Services	14-25
14.7.1	Define a LIST Service	14-25
14.7.1.1	Example: Define a LIST Service	14-25
14.7.1.2	Example: Add to a LIST Service	14-26
14.7.1.3	Example: Invoke a LIST Service	14-26
14.7.1.4	Example: Inquire on Registrations of a LIST Service	14-26
14.7.2	LIST Service's Default Function	14-27
14.8	Manipulate ILIST Services	14-27
14.8.1	Define an ILIST Service	14-27
14.8.1.1	Example: Define an ILIST Service	14-28
14.8.1.2	Example: Add to an ILIST Service	14-28
14.8.1.3	Example: Invoke an ILIST Service	14-29
14.8.2	ILIST Service's Default Function	14-30
14.9	Manipulate PID_LIST Services	14-30

14.9.1	Define a PID_LIST Service	14-31
14.9.1.1	Example: Define a PID_LIST Service	14-31
14.9.1.2	Example: Add to a PID_LIST Service	14-32
14.9.1.3	Example: Invoke a PID_LIST Service	14-32
14.9.2	PID_LIST Service's Default Function	14-32
14.10	Manipulate CASE Services	14-32
14.10.1	Define a CASE Service	14-33
14.10.1.1	Example: Define a CASE Service	14-33
14.10.1.2	Example: Add a CASE Service	14-34
14.10.1.3	Example: Inquire on Registrations of a Particular CASE Value	14-35
14.10.1.4	Example: Invoke a CASE Service	14-35
14.10.2	CASE Service's Default Function	14-35
14.10.2.1	Example: Add a Default Case Function	14-36
14.11	Manipulate RETVAL Services	14-36
14.11.1	Example 1: TRUE/FALSE Return Value with Indexed Array Structure	14-36
14.11.2	Example 2: Return of Pointer, Smaller Non-indexed Array	14-37
14.11.3	RETVAL Service's Default Function	14-38
14.12	Manipulate FASTCASE Services	14-38
14.12.1	Define a FASTCASE Service	14-38
14.12.1.1	Example: Define a FASTCASE Service	14-39
14.12.1.2	Example: Add a FASTCASE Service	14-39
14.12.1.3	Example: Invoke a FASTCASE Service	14-39
14.12.2	FASTCASE Service's Default Function	14-40
14.13	Manipulate LOOP Services	14-40
14.13.1	Define a LOOP Service	14-40
14.13.1.1	Example: Define a LOOP Service	14-41
14.13.1.2	Example: Add to a LOOP Service	14-41
14.13.1.3	Example: Invoke a LOOP Service	14-42
14.13.2	LOOP Service's Default Function	14-42
14.14	Manipulate STUB Services	14-42
14.14.1	Define a STUB Service	14-43
14.14.1.1	Example: Define a STUB Service	14-43
14.14.1.2	Example: Add to a STUB Service	14-44
14.14.1.3	Example: Invoke a STUB Service	14-44
14.14.1.4	Example: Inquire on Registrations of a STUB Service	14-44
14.14.2	STUB Service's Default Function	14-44
14.15	Manipulate STUB_CHK Services	14-45
14.15.1	Define a STUB_CHK Service	14-45
14.15.1.1	Example: Define a STUB_CHK Service	14-46
14.15.1.2	Example: Add to a STUB_CHK Service	14-46
14.15.1.3	Example: Invoke a STUB_CHK Service	14-47
14.15.2	STUB_CHK Service's Default Function	14-47
14.16	Manipulate FASTSTUB Services	14-47
14.16.1	Define a FASTSTUB Service	14-48
14.16.1.1	Example: Define a FASTSTUB Service	14-48
14.16.1.2	Example: Add a FASTSTUB Service	14-50
14.16.1.3	Example: Invoke a FASTSTUB Service	14-51
14.16.2	FASTSTUB Service's Default Function	14-51
14.17	Manipulate VALUE Services	14-52
14.17.1	Define a VALUE Service	14-52

14.17.1.1	Example: Define a VALUE Service	14-52
14.17.1.2	Example: Add to a VALUE Service	14-53
14.17.1.3	Example: Invoke a VALUE Service	14-53
14.17.2	VALUE Service's Default Function	14-53
14.17.2.1	Example: Add a Default Value	14-54
14.18	Manipulate CASE_LIST/CASE_LOOP Services	14-54
14.18.1	API Changes	14-56
14.18.1.1	Add a Callback Function to the Default List of Callbacks	14-56
14.18.1.2	Add a Callback Function to the Fallback List of Callbacks	14-56
14.18.1.3	Add a Callback Function to the Main List	14-57
14.18.1.4	Delete Callbacks from the Main List	14-57
14.18.1.5	Delete All Callbacks in the Default List	14-57
14.18.1.6	Delete Callback Function from Default List	14-57
14.18.1.7	Delete All Callbacks from the Fallback List	14-57
14.18.1.8	Delete the Callback Function from the Fallback List	14-57
14.18.1.9	Delete the Callback Function from the Main List	14-58
14.18.1.10	Invoke the Registry Service Name for Specified Tag	14-58
14.18.1.11	Check if Specified Tag is Currently Being Used	14-58
14.19	Manipulate SEQ_LIST Services	14-58

Chapter 15 Time-of-Day Services 15-1

15.1	Overview: Time-of-Day Services	15-1
15.1.1	Epoch: Definition	15-1
15.1.2	Time Formats	15-1
15.1.2.1	clock_epoch Structure	15-2
15.1.2.2	UNIX Format	15-2
15.1.2.3	ios_timeval Structure	15-2
15.1.3	System Clock: Description	15-2
15.1.4	Time Zones	15-3
15.1.5	Network Time Protocol	15-3
15.1.6	System Clock Changes	15-3
15.1.7	Hardware Calendar	15-3
15.2	Get the Current Time	15-4
15.3	Test for Summer Time	15-4
15.4	Convert between Time Formats	15-5
15.5	Set the System Clock	15-5
15.6	Determine Validity of System Clock Time	15-5
15.7	Format Time Strings	15-5

Chapter 16 Timer Services 16-1

16.1	Overview: Timer Services	16-1
16.1.1	System Clock	16-2
16.1.2	Implementing Application-Level Functions	16-2
16.1.3	Timer Jitter	16-2
16.2	Timer States	16-2
16.3	Passive Timers	16-3
16.3.1	Passive Timer Delay Descriptors	16-3
16.3.2	Passive Timers in the Future	16-3
16.3.2.1	Operation of Passive Timers in the Future	16-3

16.3.2.2	Start a Passive Timer in the Future	16-4
16.3.2.3	Set the Expiration for a Passive Timer	16-4
16.3.2.4	Stop a Passive Timer in the Future	16-4
16.3.2.5	Determine the State of Passive Timers in the Future	16-5
16.3.2.6	Guidelines for Using the SLEEPING and AWAKE Macros in Releases Prior to Release 11.1	16-5
16.3.2.7	Guidelines for Using the XSLEEPING and XAWAKE Macros in Releases Prior to Release 11.1	16-5
16.3.2.8	Guidelines for Avoiding Timer Ambiguity	16-6
16.3.2.9	Determine the Earlier of Two Timers	16-6
16.3.2.10	Compare Passive Timers in the Future	16-6
16.3.2.11	Update Passive Timers in the Future	16-6
16.3.2.12	Use One Timer Value to Compute Another	16-7
16.3.2.13	Example: Passive Timers in the Future	16-7
16.3.3	Watching a Single Passive Timer	16-8
16.3.3.1	Example: Watching a Single Passive Timer	16-8
16.3.4	Passive Timers in the Past	16-8
16.3.4.1	Determine the Current Time	16-9
16.3.4.2	Copy a Timestamp	16-9
16.3.4.3	Determine the Elapsed Time	16-9
16.3.4.4	Determine Whether a Time Is within a Range	16-9
16.3.4.5	Example: Passive Timers in the Past	16-10
16.3.5	Compare Timestamps	16-10
16.4	Managed Timers	16-11
16.4.1	Overview: Managed Timers	16-11
16.4.2	Type and Context Values	16-11
16.4.3	Recursive Managed Timers	16-11
16.4.4	Operation of Managed Timers	16-11
16.4.4.1	Using Interrupt Routines and Managed Timers	16-11
16.4.4.2	Active, Stopped, and Expired Managed Timers	16-12
16.4.5	mgd_timer Data Structure	16-12
16.4.6	Guidelines for Using Managed Timers	16-12
16.4.7	Initialize Managed Timers	16-13
16.4.8	Determine Initialization Status of a Managed Timer	16-13
16.4.9	Modify the Timer Type	16-13
16.4.10	Modify the Timer Context	16-14
16.4.11	Start a Leaf Timer	16-14
16.4.12	Increase the Delay of a Leaf Timer	16-14
16.4.13	Set a Leaf Timer's Expiration	16-14
16.4.14	Stop a Managed Timer	16-15
16.4.15	Determine the State of a Managed Timer	16-15
16.4.16	Esoteric Managed Timer Functions	16-16
16.4.16.1	Link and Delink Timer Trees	16-16
16.4.16.2	Set Extended Context	16-16
16.4.16.3	Create Fenced Timers	16-16
16.4.16.4	Convert Timers	16-17
16.4.16.5	Traverse a Tree of Managed Timers	16-17
16.4.17	Example: Managed Timers	16-17
16.5	Timer Wheel Timers	16-19
16.5.1	Timer Wheel Terms	16-20
16.5.2	Timer Wheel Timers Background	16-20
16.5.3	Timer Wheel Timers Benefits	16-24
16.5.4	A Generic Timer Wheel Timer Model	16-26

16.5.4.1	Primary Timer Wheel Data Structures	16-27
16.5.5	Timer Wheel Timers API	16-27
16.5.5.1	Create and Initialize The Timer Wheel API	16-28
16.5.6	Steps To Create A Timer Wheel Timer Service	16-28
16.5.7	Enhanced Timer Wheel (<i>New in 12.4T</i>)	16-31
16.5.7.1	Enhanced Timer Wheel Implementation	16-32
16.5.7.2	Enhanced Timer Wheel API	16-32
16.6	Choose Which Type of Timer to Use	16-33
16.6.1	The Effect of Changing a Watched Timer While Waiting for Events	16-33
16.7	Determine System Uptime	16-34
Chapter 17	Strings and Character Output	17-1
17.1	Print Strings	17-1
17.1.1	Print a String to the Connected Terminal	17-1
17.1.2	Print a Debugging String	17-1
17.1.2.1	Differences between buginf() and printf()	17-2
17.1.3	Print a String into a Buffer	17-2
17.1.4	Print a String to Nonvolatile Storage	17-2
17.1.5	Turn on Automatic "---MORE---" Processing	17-2
17.1.6	Change Automore's Header in Midstream	17-3
17.1.7	Disable "---MORE---" Processing	17-3
17.1.8	Find Out if User has Quit Automore	17-3
17.1.9	Conditionally Prompt to Do More Output	17-4
17.1.10	Format Time Strings	17-4
17.1.10.1	Examples: Format Time Strings	17-5
17.1.11	Format Timestamps	17-5
17.1.11.1	Examples: Format Timestamps	17-6
17.1.12	Format AppleTalk Addresses	17-8
17.1.12.1	%a Format Code	17-8
17.1.12.2	%A Format Code	17-9
17.1.13	Format Banyan VINES Addresses	17-10
17.1.13.1	%z Format Code	17-10
17.1.13.2	%Z Format Code	17-11
17.1.14	Format IPv6 Addresses	17-12
17.1.14.1	%P Format Code	17-12
Chapter 18	Exception Handling	18-1
18.1	Overview: Exception Handling	18-1
18.2	List of Exceptions	18-1
18.3	Register an Exception Handler	18-2
18.3.1	Register a One-Time Handler	18-2
18.3.1.1	Example: Register a One-Time Handler	18-2
18.3.2	Register a Permanent Handler	18-3
18.3.2.1	Example: Register a Permanent Handler	18-3
18.4	Cause Exceptions	18-4
18.4.1	Example: Cause Exceptions	18-4
Chapter 19	Writing Cisco IOS Error Messages	19-1
19.1	Error Message Guidelines	19-2
19.1.1	What to Avoid in Error Messages	19-4
19.2	Submitting Error Messages for Review	19-4

19.3	Defining an Error Message	19-4
19.3.1	Error Message Files	19-4
19.3.1.1	msg_*.c files That Have Been Replaced by msg_*.rc Files	19-6
19.3.2	Defining Error Messages in the Error Message File	19-6
19.3.2.1	Setting up #include's and #define's	19-7
19.3.2.2	Defining an Error Message Facility	19-7
19.3.2.3	Defining Error Messages	19-7
19.3.2.4	Creating an Error Message Explanation	19-12
19.3.2.5	Specifying a Recommended Action	19-14
19.3.2.5.1	Calling TAC - The Last Resort	19-15
19.3.2.6	Specifying a DDTS Component	19-18
19.3.2.7	Identifying Information for TAC Engineer	19-18
19.4	Testing the Error Message	19-19
19.5	Generating Error Messages	19-20
19.5.1	Example of the errmsg() Function	19-20
19.5.2	Example of the errmsg() Function With Variables	19-20
19.5.3	Use errmsg_ext() in Remote Registry Calls	19-21
19.5.3.1	Adding errmsg_ext() to Remote Registry Calls	19-21
19.6	Coding Error Messages Example	19-22

Chapter 20 Debugging and Error Logging 20-1

20.1	Debugging and Error Logging Facilities Overview	20-1
20.1.1	Debugging and Error Logging Facilities Comparison	20-2
20.1.2	Event Tracing versus Debugging	20-2
20.1.2.1	How to Balance the Benefits of Event Tracing with the Costs of Event Tracing per Platform	20-8
20.1.2.2	Possible Improvements for Event Tracing	20-8
20.2	The Debug Facility and Exceptions	20-14
20.2.1	Use Core Files to Debug CPU Exceptions	20-14
20.2.1.1	Configure the Cisco IOS Software to Generate a Core File	20-16
20.2.1.2	Analyze a Core File	20-16
20.2.2	Debug with the ROM Monitor	20-16
20.2.3	Debug with GDB	20-17
20.2.3.1	Debug in GDB Kernel Mode	20-17
20.2.3.2	Debug in GDB Process Mode	20-18
20.3	Debug with buginf() and the debug Command	20-18
20.3.1	Debug Critical Code Sections	20-19
20.3.1.1	Differences between buginf() and printf()	20-19
20.4	How to Stop Buffered Debugging from Dropping buginfs()	20-20
20.5	Enable Debug commands during boot-up in IOS	20-20
20.6	Debug Using Compile-Time Conditionals or Code Features	20-20
20.6.1	Trace Buffer Leaks	20-21
20.6.1.1	Example: Trace Buffer Leaks	20-21
20.6.2	Unwedge an Input Queue Throttled Due to Buffer Leaks	20-23
20.6.2.1	Other Cisco IOS Features Used by the Interface Unwedging Feature	20-23
20.6.2.2	memory debug leak(s) reclaim Command	20-24
20.6.2.3	Configure the Interface UnWedging Feature	20-25
20.6.2.4	Interface Unwedging Configuration File Example	20-26
20.7	Links to Other Debugging Documentation	20-27

20.8	The Event Trace Facility	20-28
20.9	What is the Event Tracer?	20-28
20.9.1	Event Trace Facility Key Features	20-29
20.9.2	Event Tracer Subsystem Files	20-30
20.9.3	Event Trace Facility Restrictions and Limitations	20-31
20.10	How to Use the Event Tracer	20-31
20.10.1	Example Setup of the Event Tracer	20-31
20.10.1.1	Addition of Component ID	20-32
20.10.1.2	Framework of ipc_trace.c	20-32
20.10.1.2.1	Instantiating an Event Trace	20-32
20.10.1.2.2	Writing the Trace Hook Function	20-33
20.10.1.2.3	Writing the Pretty Format Function	20-33
20.10.1.3	Framework of ipc_trace.h	20-33
20.10.1.4	Modification of the makefile	20-33
20.11	Event Tracer API Functions	20-34
20.11.1	Initialization	20-34
20.11.2	Opening an Event Trace	20-34
20.11.3	Enabling in One-shot Mode	20-34
20.11.4	Disabling an Event Trace	20-35
20.11.5	Storing an Event	20-35
20.11.6	Displaying Event Trace Entries	20-35
20.11.6.1	Printing All the Event Traces	20-36
20.11.7	Changing the Event Trace Buffer Size	20-36
20.11.8	Clearing the Event Trace	20-37
20.11.9	Storing the Event Trace in the Filesystem	20-38
20.11.9.1	Dumping All the Event Traces into the Filesystem	20-38
20.11.10	Closing an Event Trace	20-39
20.11.11	Inserting a Control Structure of an Event Trace into a Global List	20-39
20.12	Event Tracer Data Structures	20-39
20.12.1	Examining Event Trace Data	20-40
20.13	Event Tracer CLI Commands	20-41
20.13.1	Exec Mode Commands	20-42
20.13.2	Exec Show Mode Commands	20-42
20.13.3	Config Mode Commands	20-43
20.14	What is the Enhanced Error Message Log Count	20-43
20.14.1	End User Interface	20-44
20.14.1.1	How to Enable this Feature	20-44
20.14.1.2	How to Disable this Feature	20-44
20.14.1.3	How to Verify Whether You Have this Feature or Not	20-44
20.14.1.4	Using this Feature	20-45
20.15	The Receive Latency Trace Facility	20-45
20.15.1	Overview	20-46
20.15.2	Storing Packet Information	20-46
20.15.3	The Periodic Function	20-49
20.15.4	Time Threshold Messages	20-51
20.15.5	Snapshot Display	20-52
20.16	Latency Tracer API Functions	20-52
20.17	Latency Tracer CLI Commands	20-53
20.17.1	Parser Exec Command	20-53

20.17.2	Parser Show Command	20-54
20.17.3	Parser Test Command	20-54
20.18	Latency Tracer Implementation	20-54
20.18.1	Platform Specific Server	20-55
20.18.2	Platform Specific Client	20-55
20.18.3	Implementation File List	20-56
20.19	Creating a Media RX Trace Instance	20-57
20.19.1	Media Instance Creation	20-57
20.19.1.1	Server Instance Initialization	20-57
20.19.1.2	Client Instance Initialization	20-58
20.19.1.3	Parser Instance Initialization	20-59
20.19.2	Copying Another Media Instance	20-60
20.20	Rx Trace Server and Client APIs	20-61
20.21	Traceback Recording	20-62
20.21.1	Traceback Recording List of Functions	20-63
20.21.2	Traceback Recording Implementation	20-64
20.22	Memory Traceback Recording	20-65
20.23	TCL	20-65
20.24	Embedded Syslog Manager	20-66
20.25	How to Capture Console Output	20-66
20.26	Cisco Error Number (errno)	20-67
20.26.1	Usage Guidelines	20-68
20.26.1.1	Overview	20-68
20.26.1.2	Functional Structure and Data Structures	20-68
20.26.1.3	Interface Description	20-69
20.26.1.3.1	Error Code String Mappings	20-69
20.27	Debugging ASLR Enabled Cisco IOS Images	20-72
20.27.1	ASLR and Impact on Debugging	20-72
20.27.2	Determining the RDO and RTO offset	20-73
20.27.3	Debugging ASLR enabled images	20-73
20.27.3.1	Debugging RHO and RIO	20-73
20.27.3.2	Debugging console output of RTO and RDO enabled images	20-73
20.27.3.3	Rsym modifications for ASLR	20-73
20.27.3.4	RTO, RDO, Core Dumps, and GDB	20-74
20.27.3.5	RTO, RDO and Crashinfo, and GDB	20-74
20.28	The sdec Tool and Stack Corruption Troubleshooting	20-74
20.28.1	Find Out If There Has Been Stack Corruption	20-75
20.28.2	Check the Stacks	20-75
20.28.3	Decode	20-76
20.28.4	Find Out What Causes a Stack to Overrun	20-80
20.28.5	Resolving the Stack Overrun Problem	20-81
20.28.6	Find CDETS Defects with Similar Problems	20-81
20.28.7	Find Out How Much Stack a Function is Allocating	20-82

PART 4 Network Services

Chapter 21 Binary Trees 21-1

21.1	Overview: Binary Trees	21-1
21.1.1	Red-Black (RB) Trees	21-2
21.1.2	AVL Trees	21-2
21.1.3	Radix Trees	21-2
21.2	Manipulate RB Trees	21-3
21.2.1	Initialize an RB Tree	21-3
21.2.2	Insert a Node into an RB Tree	21-3
21.2.3	Search an RB Tree	21-3
21.2.4	Apply a Function to an RB Tree Node	21-4
21.2.5	Retrieve Information about an RB Tree	21-4
21.2.6	Print the Nodes in an RB Tree	21-4
21.2.7	Protect a Node in an RB Tree	21-4
21.2.8	Place a Node on the Tree's Internal Free List	21-4
21.2.9	Remove an RB Tree	21-5
21.2.10	Set Up an RB Tree With a Key That Is Non-32 Bits	21-5
21.3	AVL Trees	21-5
21.3.1	Manipulate Raw AVL Trees	21-6
21.3.1.1	Initialize an AVL Tree	21-6
21.3.1.2	Insert a Node into an AVL Tree	21-6
21.3.1.3	Traverse an AVL Tree	21-6
21.3.1.4	Search an AVL Tree	21-7
21.3.1.5	Remove an AVL Tree	21-7
21.3.1.6	Remove a Node from an AVL Tree	21-7
21.3.1.7	Free AVL Tree Resources	21-7
21.3.2	Manipulate Wrapped AVL Trees	21-7
21.3.2.1	Using WAVL Data Structures and Defining Necessary Routines	21-8
21.3.2.2	Initialize a Wrapped AVL Tree	21-8
21.3.2.3	Insert a Node into a Wrapped AVL Tree	21-9
21.3.2.4	Traverse a Wrapped AVL Tree	21-10
21.3.2.5	Search a Wrapped AVL Tree	21-11
21.3.2.6	Remove a Node from a WAVL Tree	21-12
21.3.2.7	Reset Pointers	21-12
21.3.2.8	Free WAVL Tree Resources	21-12
21.3.3	Manipulate Threaded AVL Trees	21-12
21.3.3.1	Benefits of Right-Threaded Trees	21-13
21.3.3.2	An Example of a Right-Threaded Tree	21-13
21.3.3.3	How Is the Threaded AVL Node Protected?	21-13
21.3.3.4	Comparison with Other Trees	21-14
21.3.3.5	New AVL Node Structure	21-14
21.3.3.6	New Threaded AVL Functions	21-15
21.3.3.7	How to Use Threaded AVL Functions	21-15
21.3.4	Manipulate AVLDup Trees	21-16
21.3.4.1	Initialize an AVL Duplicate Tree	21-16
21.3.4.2	Insert a Node into an AVLDup Tree	21-16
21.3.4.3	Delete a Node in an AVLDup tree	21-16
21.3.4.4	Search an AVLDup tree	21-16
21.3.4.5	Walk an AVLDup tree	21-17
21.3.4.6	Retrieve Next AVLDup Node	21-17
21.3.4.7	Retrieve First AVLDup Node	21-17
21.4	Manipulate Radix Trees	21-17
21.4.1	Initialize a Radix Tree	21-17
21.4.2	Insert a Node into a Radix Tree	21-17

21.4.3	Traverse a Radix Tree	21-17
21.4.4	Search for a Node in a Radix Tree	21-18
21.4.5	Mark Parent Nodes in a Radix Tree	21-18
21.4.6	Delete a Node from a Radix Tree	21-18
21.5	String Database for Fast Lookup	21-19
21.5.1	Design Overview	21-19
21.5.2	API Overview	21-19
21.5.2.1	sdb_register_component()	21-19
21.5.2.2	sdb_add_string()	21-19
21.5.2.3	sdb_remove_string()	21-20
21.5.2.4	sdb_find_string()	21-20
21.5.2.5	sdb_string_addr()	21-20

Chapter 22 Queues and Lists 22-1

22.1	Overview: Queues and Lists	22-1
22.1.1	Singly Linked Lists (Queues)	22-1
22.1.2	Doubly Linked Lists	22-2
22.2	Manipulate Queues	22-2
22.2.1	Initialize a Queue	22-2
22.2.2	Determine the State of a Queue	22-3
22.2.3	Determine Whether an Item Is on a Queue	22-3
22.3	Manipulate Direct Queues	22-3
22.3.1	Manipulate Unprotected Direct Queues	22-3
22.3.1.1	Add an Item to a Queue	22-3
22.3.1.2	Remove an Item from a Queue	22-4
22.3.1.3	Examples: Manipulate Unprotected Direct Queues	22-4
22.3.2	Manipulate Protected Direct Queues	22-5
22.3.2.1	Add an Item to a Queue	22-5
22.3.2.2	Remove an Item from a Queue	22-6
22.3.2.3	Example: Manipulate Protected Direct Queues	22-6
22.4	Manipulate Indirect Queues	22-6
22.4.1	Add an Item to a Queue	22-6
22.4.2	Change the Size of a Queue	22-7
22.4.3	Iterate over Each Item in a Queue	22-7
22.4.4	Remove an Item from a Queue	22-7
22.4.5	Examples: Manipulate Indirect Queues	22-7
22.5	Manipulate Simple Doubly Linked Lists	22-8
22.5.1	Add an Item to a Doubly Linked List	22-9
22.5.2	Remove an Item from a Doubly Linked List	22-9
22.5.3	Example: Manipulate Doubly Linked Lists	22-9
22.6	Manipulate Doubly Linked Lists with the List Manager	22-10
22.6.1	Overview: List Manager	22-10
22.6.2	Create a List	22-10
22.6.3	Modify an Existing List	22-10
22.6.4	Add an Item to a List	22-11
22.6.5	Move an Item to Another List	22-11
22.6.6	Remove an Item from a List	22-11
22.6.7	Change the Behavior of List Action Vectors	22-12
22.6.8	Retrieve the Behavior of List Action Vectors	22-12
22.6.9	Display the Contents of a List	22-12
22.6.10	Destroy a List	22-13

22.6.11 Examples: Manipulate Doubly Linked Lists with the List Manager	22-13
22.7 Other Doubly Linked List Functions	22-15
22.7.1 Add Element to End of List	22-15
22.7.2 Add Element After Specified Object	22-15
22.7.3 Add Element Before Specified Object	22-16
22.7.4 Add Element to Head of List	22-16
22.7.5 Search For Element in List	22-16
22.7.6 Initialize Linked List Management Object	22-16
22.7.7 Insert Element In The Ordering Specified	22-16
22.7.8 Read Nth Element	22-16
22.7.9 Read First Element	22-16
22.7.10 Remove Element From List	22-17
22.8 Iterate a List or Queue Shared with Interrupt Level Code	22-17
22.9 Indexed Object Lists	22-17
22.9.1 Index Objects and the Comparison Function	22-17
22.9.1.1 Syntax of the Comparison Function and Index Object	22-18
22.9.2 Indexed Object APIs	22-18
22.9.2.1 Initializing an Index Object	22-18
22.9.2.2 Adding an Element to the Indexed Object List	22-19
22.9.2.3 Removing an Element From an Indexed Object List	22-19
22.9.2.4 Finding the Size of the Indexed Object List	22-19
22.9.2.5 Searching for an Element in an Indexed Object List	22-19
22.9.2.6 Returning a Pointer to the User Data at the Nth Value in the Indexed Object List	22-19
22.9.2.7 Cleaning Up Indexed Object Lists	22-20
22.9.3 Example:Indexed Object Lists	22-20
22.10 Index Tables	22-23
22.10.1 Creating an Index Table	22-23
22.10.2 Removing an Element from an Index Table	22-23
22.10.3 Freeing all Nodes of an Index Table	22-23
22.10.4 Finding the First Element of an Index Table	22-23
22.10.5 Finding the First Empty Cell of an Index Table	22-23
22.10.6 Finding the Last Element Stored in an Index Table	22-24
22.10.7 Finding the Following Element in an Index Table	22-24
22.10.8 Finding the Next Empty Cell in an Index Table	22-24
22.10.9 Getting Nth Element in an Index Table	22-24
22.10.10 Finding the Previous Element in an Index Table	22-24
22.10.11 Inserting an Element into an Index Table	22-24

Chapter 23 Switching 23-1

23.1 Switching, an Overview	23-6
23.1.1 Low-End and Mid-Range Systems	23-8
23.1.1.1 Process Switching	23-10
23.1.1.2 Fast Switching	23-11
23.1.2 7000-Specific Switching Paths	23-12
23.1.3 Switching Paths on High-End Platforms	23-13
23.1.3.1 Route Switch Processor Architecture Overview	23-14
23.1.3.2 Packet Forwarding on Non-VIP-Based Line Cards	23-15
23.1.3.2.1 Process Switching	23-16
23.1.3.2.2 Fast and Optimum Switching	23-17
23.1.3.2.3 NetFlow Switching	23-18
23.1.3.2.4 Cisco Express Forwarding (CEF)	23-19
23.1.3.3 Packet Forwarding on VIPs with Distributed Switching	23-22

23.1.3.3.1	VIP Packet Forwarding Using dCEF and Distributed Switching	23-22
23.1.3.4	Packet Forwarding on 720x Platforms	23-24
23.1.3.4.1	Packet Forwarding on 7206VXR/NPE Route Processor Cards	23-24
23.1.3.4.2	Packet Forwarding on 7200VXR / NSE-1 Processor Cards	23-24
23.1.3.5	Packet Forwarding on the ESR 10000	23-27
23.1.3.6	Gigabit Switch Router (GSR) Switching Path	23-29
23.1.3.7	Distributed Switching Compatibility Matrix	23-30
23.1.3.7.1	With CEF Globally Disabled	23-31
23.1.3.7.2	With CEF Globally Enabled	23-31
23.1.3.7.3	With dCEF Globally Enabled	23-32
23.1.4	Load Balancing and Other Features	23-32
23.1.4.1	Load Balancing along Switching Paths	23-32
23.1.4.2	CEF Enabled, Additional Features	23-33
23.1.4.2.1	CAR	23-33
23.1.4.2.2	Unicast RPF	23-34
23.1.4.3	Access Lists	23-34
23.2	Writing Fast Switching Code	23-35
23.2.1	Hardware Architecture	23-35
23.2.1.1	MCI/CiscoBus Architecture	23-35
23.2.1.2	Shared-Memory Architecture	23-41
23.2.1.2.1	Receive a Packet	23-41
23.2.1.2.2	Make the Forwarding Decision	23-42
23.2.1.2.3	Transmit a Packet	23-42
23.2.2	Software Architecture	23-43
23.2.2.1	Full Matrix	23-43
23.2.2.2	Unique Routines	23-44
23.3	Adding a CEF Feature	23-44
23.3.1	FIB IDBs	23-44
23.3.1.1	FIB IDB Subblocks	23-45
23.3.2	How FIB Technology Works	23-45
23.3.2.1	Adjacency Tables	23-46
23.3.2.1.1	Special Adjacency Types	23-47
23.3.2.2	Example Forwarding Code	23-48
23.3.2.3	Background Processes	23-48
23.3.3	Guidelines for Adding a New Feature	23-49
23.3.3.1	Performance Dos and Don'ts	23-50
23.3.3.2	Design, Code Reviews, and Questions	23-50
23.3.4	Distributed CEF (dCEF)	23-50
23.3.4.1	Load Balancing for CEF	23-51
23.3.4.1.1	Per-Destination Load Balancing	23-51
23.3.4.1.2	Per-Packet Load Balancing	23-52
23.3.4.2	Debugging	23-52
23.3.4.2.1	Show Commands	23-52
23.3.4.2.2	Debug Commands	23-53
23.3.4.3	Resources	23-54
23.4	FIB Subblocks	23-55
23.4.1	Introduction to FIB Subblocks	23-55
23.4.1.1	What Are FIB Subblocks and Why Are They Needed?	23-55
23.4.1.2	Where Is the FIB Subblock Facility Supported?	23-56
23.4.1.3	Differences Between FIB Subblocks and Normal Subblocks	23-56
23.4.1.4	Nondistributed and Distributed Environments	23-56
23.4.1.5	Example FIB Subblock Implementation	23-57

23.4.1.6	fibidbtype and fibhwidbtype Subblocks	23-57
23.4.1.7	FIB Subblock Function Table	23-58
23.4.1.8	Example: FIB Function Tables	23-59
23.4.2	Managing FIB Subblocks	23-60
23.4.2.1	FIB Subblock Data Structures	23-60
23.4.2.1.1	Creating, Initializing, and Deleting the FIB Subblock Facility	23-61
23.4.3	Using FIB Subblocks	23-63
23.4.3.1	XDR Support Routines for the Distributed Environment	23-63
23.4.3.1.1	Control Data Synchronization	23-64
23.4.3.1.2	Statistic Data Synchronization	23-65
23.4.3.1.3	Event Data Synchronization	23-66
23.4.4	Debugging Subblocks	23-67
23.4.4.1	Frequently Asked Question	23-68
23.4.4.2	Further Help and Support	23-69
23.5	Hardware Session and L2 Hardware Switching	23-69
23.5.1	Background	23-70
23.5.1.1	Architectural Models Supported by the HW Abstraction API	23-70
23.5.1.1.1	Common Definitions for Both Local Termination and L2-L2 Switching	23-71
23.5.1.1.2	Common Counter Block	23-72
23.5.1.1.3	The hw_switching_down() Callback Mechanism	23-72
23.5.2	Support for Locally-Terminated Tunnels	23-72
23.5.2.1	The Hardware Session API	23-73
23.5.2.1.1	List of Functions that Support the Hardware Session API	23-74
23.5.3	Support for Layer 2 to Layer 2 Point-to-Point Connections	23-74
23.5.3.1	The L2 Hardware Switching API	23-74
23.5.3.1.1	L2 Hardware State Machine Description	23-77
23.5.3.1.2	Adding Features to the Hardware Switch	23-79
23.5.3.1.3	List of Functions that Support the L2 Hardware Switching API	23-79
23.5.4	Segment and Session Information Handling Details	23-80
23.6	L2VPN Platform API	23-80
23.6.1	Primary Components of SAL	23-81
23.6.2	Typical Segment Events	23-84
23.6.3	Types of Segment API Functions	23-85
23.6.4	AC Segment API	23-86
23.6.4.1	Provision an AC	23-86
23.6.4.2	Update an AC	23-87
23.6.4.3	Remove an AC	23-87
23.6.4.4	Bind an AC	23-87
23.6.4.5	Unbind an AC	23-88
23.6.4.6	Adjacency Change Notification	23-88
23.6.4.7	QoS Policy Updates Notification	23-88
23.6.4.8	Other Segment Updates Notification	23-88
23.6.4.9	Counter Collection APIs	23-89
23.6.4.10	Platform Send Event Request	23-89
23.6.4.11	Retrieve AC Circuit Information	23-89
23.6.4.12	Retrieve LINK_RAW Adjacency Object	23-89
23.6.4.13	Retrieve AC Segment Information	23-90
23.6.4.14	Set and Get the Platform Private Handle for an AC Segment	23-90
23.6.4.15	Retrieve the Peer Segment Handle and Segment Type from the AC Segment Handle	23-90
23.6.4.16	Push Counters into the Segment Context	23-90
23.6.5	L2TP Segment API	23-90
23.6.5.1	Add a L2TP Session	23-91

23.6.5.2	Update a L2TP Session	23-91
23.6.5.3	Remove a L2TP Session	23-92
23.6.5.4	Bind a L2TP Session	23-92
23.6.5.5	Unbind a L2TP Session	23-92
23.6.5.6	Adjacency Change Notification	23-93
23.6.5.7	Other Segment Updated Notification	23-93
23.6.5.8	Start/Stop Punting Packets to Next Switching Level	23-93
23.6.5.9	Collect Counters on a L2TP Session	23-93
23.6.5.10	Retrieve LINK_PW_IP Adjacency Object	23-94
23.6.5.11	Return L2TP Segment Configuration	23-94
23.6.5.12	Set and Get the Platform Private Handle for a L2TP Segment	23-94
23.6.5.13	Retrieve the Peer Segment Handle and Segment Type from the L2TP Segment Handle	23-94
23.6.5.14	Return L2TP Segment Configuration	23-95
23.6.6	AToM Segment API - 12.2S MFI Version	23-95
23.6.6.1	Retrieve Peer Segment Information	23-95
23.6.6.2	Determine If Disposition/Imposition Has Controlword/Sequencing Processing	23-95
23.6.6.3	Retrieve the loadbalance Index	23-95
23.6.6.4	Retrieve the AToM Imposition OCE	23-96
23.6.6.5	Program a tag-rewrites	23-96
23.6.7	AToM Segment API - 12.0S AToM Switching Manager	23-96
23.6.7.1	Program a tag-rewrites	23-97
23.6.7.2	Retrieve Peer Segment Information	23-97
23.6.7.3	Retrieve Imposition Rewrite from AToM Segment Handle	23-97
23.6.7.4	Retrieve Disposition Rewrite from AToM Segment Handle	23-97
23.6.7.5	Set AToM Segment Private Platform Context	23-97
23.6.7.6	Get AToM Segment Private Platform Context	23-97
23.6.7.7	Retrieve SSM Segment ID from AToM Segment	23-97
23.6.8	VFI Segment API	23-98
23.6.8.1	Provision a VFI Segment	23-98
23.6.8.2	Unprovision a VFI Segment	23-99
23.6.8.3	Update a VFI Segment	23-99
23.6.8.4	Bind a VFI Segment	23-99
23.6.8.5	Unbind a VFI Segment	23-99
23.6.8.6	Compute and Reserve a Platform Resource	23-99
23.6.8.7	Retrieve VFI Segment Information from a VFI Segment Handle	23-100
23.6.8.8	Retrieve the Platform Private Handle from a VFI Segment Handle	23-100
23.6.8.9	Set the Platform Private Handle into a VFI Segment Handle	23-100
23.6.8.10	Retrieve the Segment Type of the Other Segment from a VFI Segment Handle	23-100
23.6.8.11	Retrieve the Segment Handle of the Other Segment from a VFI Segment Handle	23-100
23.6.8.12	Iterate Through all Switches Belonging to a Given VFI	23-100
23.6.9	HW API Usage Examples	23-101
23.6.9.1	AC to AToM HW API Overview	23-101
23.6.9.2	AC to AC (Local-switching) HW API Overview	23-102
23.6.9.3	AC to L2TP HW API Overview	23-103
23.6.9.4	VFI HW API Overview	23-104
23.6.9.5	Tunnel Stitching HW API Overview	23-107

Chapter 24 High Availability (HA) 24-1

24.1	Introduction	24-1
24.1.1	Terms	24-2
24.1.2	What is HA?	24-5
24.1.3	Assumptions & Requirements	24-6

24.2	Stateful Switchover (SSO)	24-7
24.2.1	Platform Hardware Model	24-8
24.2.1.1	Architectural Assumptions	24-8
24.2.1.2	SSO System Platform Architecture	24-8
24.2.1.3	RP Switchover Behavior	24-10
24.2.1.4	Line Card Behavior for SSO	24-10
24.2.1.5	RP Switchover Conditions	24-11
24.2.1.5.1	A Fault on the Active RP – Automatic Switchover	24-11
24.2.1.5.2	Active RP Declared “Dead” – Automatic Switchover	24-11
24.2.1.5.3	CLI Invoked – Manual Switchover	24-11
24.2.1.5.4	CLI Invoked – Trial Switchover	24-11
24.2.1.5.5	Criteria-based – Automatic Switchover	24-12
24.3	SSO Software Features	24-12
24.3.1	SSO Software Architectural Model	24-12
24.3.2	The Driver-Client Architectural Model	24-13
24.3.2.1	Model 1	24-13
24.3.2.2	Model 3	24-15
24.3.2.3	Model 1 vs. Model 3	24-15
24.3.3	Non HA-Aware Protocols and Features	24-16
24.3.4	Software Upgrade	24-16
24.3.4.1	Fast Software Upgrade (FSU)	24-17
24.3.4.2	Hitless Software Upgrade (HSU)	24-17
24.4	SSO Implementation Overview	24-17
24.4.1	SSO Platforms	24-17
24.4.2	SSO Software Features	24-17
24.4.2.1	Routing Protocols and Non-Stop Forwarding (NSF)	24-17
24.4.2.1.1	Cisco Express Forwarding (CEF)	24-18
24.4.2.1.2	Border Gateway Protocol (BGP)	24-19
24.4.2.1.3	Open Shortest Path First Protocol (OSPF)	24-19
24.4.2.1.4	Intermediate System-to-Intermediate System Interior Gateway Routing Protocol (IS-IS)	24-20
24.4.2.1.5	Graceful Shutdown	24-20
24.4.2.2	Routed Protocols	24-20
24.4.2.2.1	Point to Point Protocol (PPP)	24-20
24.4.2.2.2	Frame Relay (FR)	24-21
24.4.2.2.3	Asynchronous Transfer Mode (ATM)	24-21
24.4.2.3	IOS Infrastructure Components	24-22
24.4.2.3.1	Configuration Synchronization (Config Sync)	24-22
24.4.2.3.2	Parser Return Codes	24-23
24.4.2.3.3	File System and Other Configuration Issues in an SSO System	24-23
24.4.2.4	Redundancy Facility (RF)	24-24
24.4.2.5	Checkpointing Facility (CF)	24-25
24.4.2.6	IPC	24-25
24.4.2.6.1	Protocol Versioning	24-25
24.4.2.6.2	Windowing	24-25
24.4.2.6.3	Fragmentation and Reassembly (FAR)	24-26
24.4.2.7	Media Layer (a.k.a. network.c)	24-26
24.4.2.8	Coredump	24-26
24.4.2.9	Event Tracer	24-27
24.4.3	Platform Specific Support	24-27
24.4.3.1	Line Card Drivers	24-27
24.4.3.2	RF	24-27
24.4.3.3	IPC	24-28

24.4.3.4	APS Support	24-28
24.4.3.5	ATM Drivers and Line Cards	24-28
24.4.3.6	Post Switchover State Verification and Reconciliation	24-28
24.4.4	Network Management Support	24-28
24.5	Redundancy Facility	24-29
24.5.1	RF Clients	24-30
24.5.2	RF Client Message Transport	24-31
24.5.3	RF Infrastructure	24-32
24.5.3.1	Redundancy Control and Monitoring	24-32
24.5.3.2	Fault Management & Switchover Notification Input	24-33
24.5.4	Redundancy States	24-33
24.5.4.1	Initialization	24-34
24.5.4.2	Standby-Negotiation	24-34
24.5.4.3	Standby-Cold	24-35
24.5.4.4	Standby-Config	24-35
24.5.4.5	Standby-File System	24-35
24.5.4.6	Standby-Bulk	24-35
24.5.4.7	Standby-Hot	24-35
24.5.4.8	Active-Fast	24-35
24.5.4.9	Active-Drain	24-36
24.5.4.10	Active-Preconfig	24-36
24.5.4.11	Active-Postconfig	24-36
24.5.4.12	Active	24-36
24.5.4.13	Redundancy State Progression	24-36
24.5.4.14	RF Progression	24-37
24.5.4.14.1	Initialization Progression	24-38
24.5.4.14.2	Active Progression	24-38
24.5.4.14.3	Standby Progression	24-39
24.5.4.15	RF Progression Synchronization	24-39
24.5.4.16	RF Progression Event Call-Back API	24-40
24.5.5	Redundancy Status Events	24-40
24.5.5.1	RF Status Event Call-Back API	24-41
24.5.6	RF Messaging	24-41
24.5.6.1	RF Message Call-Back API	24-42
24.6	Sending Sync Messages	24-42
24.6.1	Client Message Considerations	24-43
24.6.2	Header Files	24-43
24.6.2.1	RF Include Files for Clients	24-43
24.6.2.2	RF Include Files for Supporting Platform Code	24-43
24.6.2.3	rf_client_ID_list.h	24-44
24.6.2.4	RF Message Header	24-44
24.6.3	RF APIs	24-44
24.6.4	RF History	24-45
24.6.5	Log Implementation Details	24-45
24.6.6	How Events Are Logged	24-45
24.6.7	History CLI	24-46
24.6.7.1	Log Output - From the Active	24-46
24.6.7.2	Standby Logs (After a Switchover)	24-47
24.6.8	Log Timestamps	24-47
24.6.8.1	syslog Timestamps	24-47
24.6.9	CLI Support	24-48
24.6.9.1	Exec Level Commands	24-48
24.6.9.2	Show Commands	24-48

24.6.9.3	Debug CLI	24-49
24.6.9.4	Clear CLI	24-50
24.6.9.5	Configure CLI	24-50
24.6.10	Platform-Dependent Support Routines	24-50
24.6.10.1	Platform Buffer Management	24-50
24.6.10.1.1	Buffer Routine Registration	24-52
24.6.10.2	Peer Messaging	24-52
24.6.10.2.1	Peer Messaging Registration	24-52
24.6.10.2.2	Open Peer Communication Registration	24-55
24.6.10.3	OK to SWACT	24-55
24.6.10.4	OK to SWACT Registration	24-55
24.6.10.5	OK to Split	24-55
24.6.10.6	OK to Split Registration	24-56
24.6.10.7	Standby Initialization	24-56
24.6.10.8	Process Configuration	24-57
24.6.10.9	Reload	24-57
24.6.10.9.1	Reload Registration	24-58
24.6.10.10	Primary Unit	24-58
24.6.11	RF MIB	24-58
24.7	Target Platform Architectures	24-58
24.7.1	Cisco 10000 ESR	24-58
24.7.1.1	Cisco 10000 ESR Forwarding and Control Architecture	24-59
24.7.2	Cisco 7500 Overview	24-60
24.7.2.1	Cisco 7500 Forwarding and Control Architecture	24-61
24.7.3	Cisco 12000 GSR Overview	24-62
24.7.3.1	Cisco 12000 GSR Forwarding and Control Architecture	24-63

Chapter 25 IP Services 25-1

25.1	IP API Functions	25-1
25.2	BEEP	25-2
25.2.1	Terms	25-2
25.2.2	The BEEPCORE-C API	25-2
25.2.3	How to Use the BEEPCORE-C API	25-3
25.2.3.1	Authoring New BEEP Profiles	25-3
25.2.3.2	Using Existing BEEP Profiles	25-3
25.3	DHCP	25-4
25.3.1	DHCP Proxy Client API	25-4
25.3.2	Using the DHCP Client Calls	25-5
25.4	DNS	25-7
25.4.1	DNS API	25-7
25.4.2	Using the DNS non-blocking API	25-8

PART 5 Hardware-Specific Design

Chapter 26 Porting Cisco IOS Software to a New Platform 26-1

26.1	Portability Issues	26-1
26.1.1	Byte Order	26-2
26.1.1.1	Unions	26-2
26.1.1.2	Bit Fields	26-3
26.1.1.2.1	Structure Padding	26-4
26.1.1.2.2	Unary Types	26-5

26.1.1.3	Bit Operations	26-5
26.1.1.4	Typecasting	26-5
26.1.2	Data Alignment	26-6
26.1.2.1	MC68000 Memory Addressing Examples	26-6
26.1.3	Data Size	26-6
26.1.4	Adding a New CPU Type	26-7
26.1.5	Other Portability Issues	26-8
26.1.5.1	Performance	26-8
26.1.5.2	Stack Usage and Stack Growth	26-9
26.1.5.2.1	Reasons for Stack Overflow	26-10
26.1.5.3	Register Considerations	26-10
26.1.5.4	Compliance with Encapsulations	26-10
26.2	Cisco's Implementation of Portability	26-12
26.2.1	Inline Assembler	26-12
26.2.2	Header Files	26-12
26.2.3	Byte-Order Functions	26-13
26.2.4	Endian #defines	26-13
26.2.5	GET and PUT Macros	26-14
26.2.6	Canonical Functions	26-14

PART 6 Management Services

Chapter 27 Command-Line Parser 27-1

27.1	Parser Information Sources	27-1
27.1.1	Using CLI Editing Features and Shortcuts	27-2
27.2	Overview: Parser	27-2
27.2.1	Traversing the Parse Tree	27-4
27.2.1.1	Parsing Config versus Commands	27-5
27.2.2	Transition Structure	27-6
27.3	Building Parse Trees	27-6
27.3.1	Construction of Parse Trees	27-7
27.3.1.1	Example: Construction of Parse Trees	27-7
27.3.1.2	Parser Chains	27-7
27.3.1.3	Maintaining Backward Compatibility	27-8
27.3.1.4	How to Avoid Collateral OBJ Damage in the Parse Trees	27-9
27.3.2	Parse a Keyword Token	27-10
27.3.2.1	Example: Parse a Keyword Token	27-12
27.3.2.2	How to Write Commands for Subinterfaces	27-13
27.3.3	Parse a Number Token	27-13
27.3.3.1	Octal Number Parsing in NUMBER and Other Related Macros	27-15
27.3.3.2	Example: Parse a Number Token	27-16
27.3.3.3	Example: Parse a Number from a Discontiguous Range	27-16
27.3.4	Parse a Keyword-Number Combination	27-17
27.3.4.1	Examples: Parse a Keyword-Number Combination	27-17
27.3.5	Parse Optional Keywords	27-18
27.3.6	Parse Mixed String and Nonstring Tokens	27-18
27.3.6.1	Example: Parse Mixed String and Nonstring Tokens	27-19
27.3.7	Negating Commands — the “No” and “Default” Keywords	27-20
27.3.7.1	“No” versus “Default”	27-20
27.3.7.2	Handling the Negative Forms of Commands	27-22
27.3.7.3	Difference between NONE and no_alt	27-23

27.3.7.4	Parameter Elision in Negative Forms - the NOPREFIX Macro	27-23
27.3.8	Nonvolatile Output Generation	27-24
27.3.9	How to Block Unsupported Commands	27-24
27.3.9.1	Definitions of "Unsupported"	27-24
27.3.9.2	Best Practices	27-24
27.3.9.3	Images	27-25
27.3.9.4	Subsystems	27-25
27.3.9.5	Parser	27-26
27.3.9.6	Registry	27-28
27.3.9.7	Command Line	27-33
27.3.9.8	PRIV_HIDDEN	27-35
27.3.9.9	PRIV_UNSUPPORTED	27-35
27.3.9.10	Documentation	27-36
27.3.10	Generating Error Messages in Configuration Mode	27-36
27.3.11	Configuration Editor in IOS?	27-37
27.4	Adding Parser Return Codes	27-37
27.4.1	PRC Overview	27-37
27.4.2	Terms	27-38
27.4.3	Functional Structure	27-39
27.4.3.1	How the PRC is Set	27-39
27.4.3.2	Users of PRC	27-39
27.4.3.3	Partial Success is Complete Failure	27-39
27.4.3.4	Successful Action Function	27-39
27.4.4	End User Interface	27-39
27.4.5	Restrictions and Configuration Issues with PRC	27-40
27.5	Hot ICE	27-40
27.5.1	Macro Modifications	27-41
27.5.2	PRC Phase III and Action Function Return Codes	27-41
27.5.2.1	Error Codes	27-42
27.5.2.2	Failure Types	27-43
27.5.2.3	Change Types	27-43
27.5.2.4	Change Modes	27-43
27.5.2.5	PRC Phase III API	27-44
27.5.2.6	New PRC Phase III AF (Action Function) Return Codes	27-44
27.5.2.7	Sample Using PRC Phase III	27-45
27.5.2.8	Migration from PRC Phase I and II	27-48
27.5.2.9	Example with Verification	27-49
27.5.3	Configuration Defaults Exposure	27-50
27.5.3.1	Configuration Defaults Exposure API	27-50
27.5.3.2	Sample Using Configuration Defaults Exposure	27-50
27.5.3.3	Example with Verification	27-52
27.5.4	Rollback Component Conformance	27-52
27.5.4.1	The Config Rollback API	27-53
27.5.4.2	Example with Verification	27-53
27.5.5	Support for Syntax Checking	27-54
27.5.5.1	Issues with Syntax Checking	27-54
27.5.5.2	How to Support Syntax Checking	27-54
27.5.5.2.1	CLI for Syntax Checking	27-55
27.5.5.3	Examples with Verification	27-56
27.5.6	Configuration Change Notification	27-58
27.5.7	Before After API	27-63
27.5.7.1	Config Data that is Changed by Others (Side Effects)	27-64
27.5.8	How to Instrument IOS Components for Hot ICE	27-64

27.5.9	Hot ICE Compliance Tooling	27-64
27.6	Linking Parse Trees	27-64
27.6.1	Example: Link Parse Trees	27-64
27.7	Manipulating CSB Objects	27-65
27.7.1	Overview: CSB Objects	27-65
27.7.2	Examples of CSB Objects	27-66
27.7.3	Important Notes on Data Variables	27-66
27.7.3.1	Background of Parsing Flaw	27-67
27.8	Ordering Commands	27-68
27.8.1	Linking Commands	27-69
27.9	Adding Commands Dynamically	27-70
27.9.1	Create a Link Point	27-70
27.9.1.1	Example: Create a Link Point	27-70
27.9.2	Register a Link Point with the Parser	27-71
27.9.2.1	Example: Register a Link Point with the Parser	27-71
27.9.3	Display Registered Link Points	27-71
27.9.4	Link Commands to a Link Point	27-72
27.9.4.1	Example: Link Commands to a Link Point	27-72
27.9.5	Create Link Exit Points	27-72
27.9.5.1	Example: Create Link Exit Points	27-73
27.10	Useful Parser Commands	27-73
27.11	Guidelines for Internal and Hidden Commands and More	27-74
27.11.1	Use Internal Commands, Not Hidden	27-74
27.11.2	One-Off Commands Treated as Normal	27-74
27.11.3	Why Hidden Commands are Discouraged	27-74
27.11.4	Guidelines for Confirmations	27-74
27.11.5	Permanently Hidden Commands	27-75
27.12	Useful Exit Functions and Submode Attributes	27-75
27.12.1	Submode Exit Functions	27-75
27.12.2	Exiting a Submode	27-75
27.12.3	Implicit Submode Exit to Parent Mode	27-76
27.12.4	The exit Command from a Submode	27-77
27.12.5	Interface Submodes and Interface Ranges	27-77
27.12.6	Limiting Parser Command Searches in Interface Submodes	27-78
27.13	Manipulating Parser Modes	27-78
27.13.1	Add a Parser Mode	27-78
27.13.1.1	Example: Add a Parser Mode	27-79
27.13.2	Add an Alias to a Mode	27-79
27.13.2.1	Example: Add an Alias to a Mode	27-79
27.14	CLI Safely Using Data Shared with Interrupt Level Code	27-79
27.15	Debugging Parser Ambiguity	27-80
27.15.1	Content of Debug Parser Ambiguity Output	27-80
27.15.1.1	Actions Performed When a no_alt node or EOL Is Encountered	27-81
27.15.1.2	Example Analysis of the Output of debug parser ambig	27-82
27.15.1.3	Identification of Parser Chain Breakage From the Output of debug parser ambig	27-84
27.15.1.3.1	Example Parser Chain Breakage	27-84
27.15.1.4	When to Use the Output of debug parser ambig	27-88
27.16	Adding New Interface and Controller Types	27-88

27.16.1	List of Macros and Functions	27-88
27.16.2	Using DEFINE_IFTYPE to Add New Interface Types	27-89
27.16.3	Using DEFINE_CFTYPE to Add New Controller Types	27-90
27.17	ICD (Intelligent Config Diff)	27-90
27.17.1	The ICD CLI	27-91
27.17.2	Providing ICD Support to Applications	27-92
27.18	IOS Config Rollback	27-93
27.18.1	The IOS Config Rollback CLI	27-94
27.18.2	Providing IOS Config Rollback Support to Applications	27-94
27.19	How to Find a Command's EOL	27-95
27.19.1	Overview	27-95
27.19.2	Rebuild IOS Image With One Minor Change	27-95
27.19.3	Find a Command's EOL node	27-96
27.20	NVGEN Enhancements	27-98
27.20.1	Config Checkpointing	27-98
27.20.1.1	NVGEN Config Checkpointing	27-98
27.21	Warning Against an Interactive CLI	27-99

Chapter 28 Writing, Testing, and Publishing MIBs 28-1

28.1	SNMP Overview	28-1
28.1.1	Internet Network Management Framework: Definition	28-2
28.1.2	MIB: Definition	28-2
28.1.3	ASN.1: Definition	28-2
28.1.4	SMI: Definition	28-2
28.1.5	Transport Protocols	28-3
28.1.6	SNMP Facilities	28-3
28.1.7	Asynchronous Notifications	28-3
28.2	MIB Concepts	28-3
28.2.1	MIB: Overview	28-3
28.2.2	Standard and Enterprise MIBs	28-4
28.2.3	MIB-I and MIB-II	28-4
28.2.4	Agent Implementations	28-4
28.2.5	MIB Objects	28-4
28.2.5.1	Object: Definition	28-4
28.2.5.2	Lexicographic Ordering of Objects	28-5
28.2.5.3	Object Identifier: Definition	28-5
28.2.6	SNMP Conceptual Tables	28-6
28.2.6.1	SNMP Conceptual Tables: Definition	28-6
28.2.6.2	Simple SNMP Conceptual Tables	28-6
28.2.6.3	Complex SNMP Conceptual Tables	28-7
28.2.6.4	Coding Index Objects	28-7
28.2.6.5	Tables Inside of Tables	28-7
28.3	SMI Overview	28-8
28.3.1	Primitive Data and Application Types	28-8
28.3.2	Textual Conventions	28-9
28.4	MIB Life Cycle	28-10
28.5	Design a MIB	28-11
28.5.1	MIB Design: Overview	28-11
28.5.2	SNMP Application Considerations	28-11

28.5.3	MIB Design Phases	28-12
28.5.3.1	Design the MIB Content	28-12
28.5.3.2	Design the Notifications	28-12
28.5.3.3	Design the MIB Organization	28-14
28.5.4	Check for Existing MIB Implementations	28-14
28.5.5	Ensure MIB Compliance	28-14
28.5.6	Follow MIB Conventions	28-14
28.5.6.1	Assigned Numbers	28-15
28.5.6.2	Conventions for Writing MIBs	28-15
28.5.7	MIB Compilers	28-20
28.5.7.1	Function of MIB Compilers	28-20
28.5.7.2	Available Compilers	28-21
28.5.7.3	Invoke the MIB Compiler	28-21
28.5.8	Agent Development	28-21
28.5.9	Cisco Internal MIB Design Support	28-22
28.6	MIB Review	28-22
28.6.1	Overview of the MIB Review Process	28-22
28.6.1.1	Tier One	28-22
28.6.1.2	Tier Two	28-22
28.6.1.3	Type of Review	28-22
28.6.2	MIB Submission Process	28-23
28.6.3	Reviewer and Approver Assignment	28-23
28.6.4	Approval Process	28-23
28.6.5	Process Exceptions	28-23
28.6.6	Policy on Acquisition MIBs	28-24
28.6.7	Rejected, Withdrawn, and Dormant MIBs	28-24
28.7	MIB Development Process: Overview	28-24
28.8	Establish a New MIB	28-25
28.9	Compile a MIB	28-28
28.9.1	Which MIB or MIBs to Compile	28-28
28.9.2	Which Groups to Compile	28-28
28.9.3	Where to Place Files Generated by the MIB Compiler	28-28
28.9.4	Makefile Rules for Compiling MIBs	28-29
28.9.5	Invoke the MIB Compiler	28-29
28.9.6	What the MIB Compiler Does	28-30
28.9.7	Output from the MIB Compiler	28-31
28.9.8	Compile a MIB: Examples	28-31
28.10	Observe Modularity	28-33
28.10.1	Subsystem	28-33
28.10.2	Instrumentation	28-33
28.11	Implement MIB Objects	28-33
28.11.1	GCC Warnings	28-33
28.11.2	Validation	28-34
28.11.3	k_get Routines	28-34
28.11.4	k_set Routines	28-35
28.12	Implement SNMP Asynchronous Notifications	28-35
28.12.1	Decide Where to Place SNMP Notification Code	28-35
28.12.2	Define the Notification	28-35
28.12.3	Control the Notification	28-36
28.12.4	Generate the Notification	28-38

28.13	Test a MIB	28-40
28.13.1	Test an Object	28-40
28.13.2	Test a Notification	28-41
28.13.3	Tools for Testing a MIB	28-41
28.13.3.1	Command-Line Tools	28-41
28.13.3.2	X Windows Tools	28-41
28.13.3.3	Notification Tools	28-42
28.13.4	SNMP Operations	28-42
28.13.5	Object Functions	28-43
28.14	Release a MIB	28-43
28.14.1	Release MIB Code	28-43
28.14.2	Release MIB Files	28-43
28.15	Maintain a MIB	28-43
28.15.1	Use MIB Versions	28-44
28.16	Testing and Publishing a MIB	28-46
28.16.1	Create or Update a MIB Workspace	28-46
28.16.2	Test a MIB	28-47
28.16.3	Analyze Test Results	28-48
28.16.4	Determine Whether You Have an SNMPv1 or SNMPv2 MIB	28-48
28.16.5	Generate an SNMPv1 Version of an SNMPv2 MIB	28-48
28.16.6	Use Make Directly to Generate a MIB	28-49
28.16.7	Use Make Directly to Generate an SNMPv2 MIB	28-49
28.16.8	Use Make Directly to Generate an SNMPv1 MIB	28-49
28.16.8.1	Example: Use Make to Generate an SNMPv1 MIB	28-49
28.16.9	Publish a MIB	28-50
28.16.10	Prerequisites for Publishing a MIB	28-50
28.16.11	MIB-Related Files	28-50
28.16.12	File Locations	28-50
28.16.13	MIB Repository and Workspace	28-51
28.16.14	Files in the MIB Repository and Workspace	28-51
28.16.15	Directory Layout for MIB Repository and Workspace	28-51

Chapter 29 MIB Infrastructure 29-1

29.1	Overview	29-1
29.1.1	Terminology	29-1
29.2	Interface MIBs Infrastructure Overview	29-3
29.2.1	Interface Manager Infrastructure	29-3
29.2.1.1	Background	29-3
29.2.1.2	Interface Manager	29-4
29.2.1.2.1	IM Architecture	29-4
29.2.1.2.2	IM Concepts	29-5
29.2.1.2.3	Type Definitions	29-7
29.2.1.2.4	Driver Registry	29-7
29.2.1.3	Interface Driver	29-8
29.2.1.3.1	Interface Driver Initialization	29-9
29.2.1.3.2	Interface Creation	29-11
29.2.1.3.3	Interface Destruction	29-13
29.2.1.3.4	Interface Get Data	29-14
29.2.1.3.5	Interface Validate Data	29-15
29.2.1.3.6	Interface Set Data	29-16
29.2.1.3.7	Interface Receive Address Data	29-18
29.2.1.3.8	Interface Stack Relationships	29-19

29.2.1.3.9	Interface Event Notification	29-20
29.2.1.4	Development Unit Testing	29-21
29.2.1.4.1	Driver Registration	29-21
29.2.1.4.2	Driver Creation/Destruction	29-22
29.2.1.4.3	ifTable/ifXTable	29-22
29.2.1.4.4	ifStackTable	29-22
29.2.1.4.5	ifAddressTable	29-22
29.2.1.4.6	IF Event Notification	29-22
29.2.2	Ethernet Interface Manager	29-22
29.2.2.1	Ethernet Interface Driver	29-23
29.2.2.1.1	Ethernet Interface Driver Initialization	29-23
29.2.2.1.2	Ethernet Interface Get Data	29-24
29.2.2.1.3	Ethernet Interface Validate Data	29-28
29.2.2.1.4	Ethernet Interface Set Data	29-29
29.2.3	IF-MIB Infrastructure	29-29
29.2.3.1	Supporting Main Interfaces, Controllers, and Subinterfaces	29-30
29.2.3.1.1	IF_MIB Tables	29-30
29.2.3.1.2	IF-MIB API	29-30
29.2.3.2	Adding Support to Register or Deregister with IF-MIB	29-31
29.2.3.2.1	Registering Main Interfaces	29-31
29.2.3.2.2	Deregistering Main Interfaces	29-31
29.2.3.2.3	Registering Controllers	29-32
29.2.3.2.4	Deregistering Controllers	29-32
29.2.3.2.5	Registering and Deregistering Sublayers	29-32
29.2.3.2.6	Adding to Service Points for Sublayers	29-32
29.2.3.2.7	Registering a Sublayer	29-34
29.2.3.2.8	Deregistering a Sublayer	29-35
29.2.3.2.9	Modifying the ifRcvAddressTable for Sublayers	29-35
29.2.3.2.10	Modifying the ifStackTable for Sublayers	29-35
29.2.3.2.11	Sparse Table Support for Sublayers	29-35
29.2.3.3	Sample Implementation: Frame Relay Sublayers	29-35
29.2.3.3.1	Adding Service Points: Frame Relay	29-35
29.2.3.3.2	Registering a Sublayer: Frame Relay	29-36
29.2.3.3.3	Deregistering a Sublayer: Frame Relay	29-36
29.2.3.4	Link Up/Down Notification Support	29-36
29.3	Entity MIB Infrastructure	29-36
29.3.1	Entity MIB API	29-37
29.3.1.1	Functions to Add Entity Objects to the ENTITY-MIB	29-37
29.3.1.1.1	Adding a Physical Entity to the ENTITY-MIB	29-37
29.3.1.1.2	Adding a Logical Entity to the ENTITY-MIB	29-37
29.3.1.1.3	Adding Logical Entity - Physical Entity IP Mapping	29-37
29.3.1.1.4	Adding Logical and Physical Entity Mapping Alias	29-37
29.3.1.2	Functions to Delete Entity Objects from the ENTITY-MIB	29-37
29.3.1.2.1	Deleting a Physical Entity from the ENTITY-MIB	29-37
29.3.1.2.2	Deleting a Logical Entity from the ENTITY-MIB	29-38
29.3.1.2.3	Deleting a Physical and Logical Entity Mapping from the ENTITY-MIB	29-38
29.3.1.2.4	Deleting a Logical and Physical Entity Mapping Alias	29-38
29.3.1.3	Functions to Look Up Entity Objects in the ENTITY-MIB	29-38
29.3.1.3.1	Looking Up Physical Entity in the ENTITY-MIB	29-38
29.3.1.3.2	Looking Up Logical Entity in the ENTITY-MIB	29-38
29.3.1.3.3	Looking Up the IP Mapping in the ENTITY-MIB	29-38
29.3.1.3.4	Looking Up the Alias in the ENTITY-MIB	29-38
29.3.1.3.5	Testing the Physical Entity in the ENTITY-MIB	29-39

29.3.1.4	Utility Functions	29-39
29.3.1.4.1	Finding the Vendor OID for a Particular HWIDB Interface	29-39
29.3.1.4.2	Formatting IP Transport Information into a String	29-39
29.3.1.5	Extension ENTITY-MIB Support	29-39
29.3.1.5.1	Adding a Physical Entity Extension Entry	29-39
29.3.1.5.2	Deleting a Physical Entity Extension Entry	29-39
29.3.1.5.3	Looking Up a Physical Extension Entry	29-39
29.4	MIB Persistence Infrastructure	29-40
29.4.1	MIB Persistence API	29-40
29.4.1.1	Writing Data to NVRAM	29-40
29.4.1.2	Loading Persistent Data	29-40
29.4.1.3	Opening Underlying File in Non-volatile Storage Area	29-40
29.4.1.4	Closing Persistence Storage	29-41
29.4.1.5	Deleting the Underlying File	29-41
29.4.1.6	Persistence Support for MIBs	29-41
29.5	SNMP Notification Infrastructure	29-41
29.5.1	SNMP Notification API	29-41
29.5.1.1	isGeneralInfoGroupOnly()	29-41
29.5.1.2	ifType_get()	29-42
29.5.1.3	reg_add_hc_hwcounter_get()	29-42
29.5.1.4	reg_add_ifmib_get_operstatus_hwidb()	29-42
29.5.1.5	reg_add_ifmib_hwifSpecific_get()	29-42
29.5.2	SNMP Notification Implementation	29-42
29.5.2.1	Flow of the Trap Generation	29-43
29.5.2.2	Flow of the Informs	29-44
29.6	HA MIB Sync Infrastructure	29-44
29.6.1	Uninterrupted SNMP View to NMS	29-45
29.6.1.1	Maintaining sysUpTime across Switchover	29-45
29.6.1.2	sysObjectID	29-46
29.6.1.3	SNMP Configuration Across Switchover	29-46
29.6.1.4	IF-MIB and ENTITY-MIB	29-46
29.6.2	Communication to the NMS	29-47
29.6.2.1	Counters/Statistics	29-47
29.6.2.2	CISCO-RF-MIB	29-47
29.6.2.3	failoverTime/redundantSysUpTime	29-49
29.6.2.4	Switchover Notification	29-49
29.6.3	General Issues Regarding SSO Functionality	29-49
29.6.3.1	Notifications Generated on the Standby	29-50
29.6.3.2	Tracking Redundancy and Switchover Information	29-50
29.6.3.2.1	The Processor Table	29-51
29.6.3.2.2	The History Table	29-51
29.6.3.3	Issues Not Addressed	29-52
29.6.3.3.1	Distributed Management MIBs	29-53
29.6.3.3.2	Categorization of Notifications (Not Considered for IOS SNMP SSO)	29-53
29.6.3.3.3	Other MIBs	29-53
29.6.3.4	Memory and Performance Impact	29-53
29.6.3.5	Packaging Considerations	29-53
29.6.4	CLI End User Interface	29-54
29.6.5	Configuration and Restrictions	29-54
29.6.6	Testing Considerations	29-54
29.6.7	SSO Application Notification	29-55
29.6.8	HA MIB Sync Infrastructure Implementation	29-55

29.7 References 29-57

Chapter 30 Security 30-1

- 30.1 Secure Coding 30-1
 - 30.1.1 How to Avoid Common Mistakes with Strings 30-2
 - 30.1.1.1 Examples 30-2
 - 30.1.2 printf() Pitfall 30-7
 - 30.1.2.1 printf(), snprintf(), and buginf() Format Strings 30-7
 - 30.1.2.2 Recommendation: Use puts() 30-8
 - 30.1.3 IOS Parser and Secure Coding 30-9
 - 30.1.4 strcpy(), strcat(), and sprintf() Limitations 30-9
 - 30.1.5 Zeroing Passwords and Keys 30-11
 - 30.1.6 Asking and Verifying Passwords 30-12
 - 30.1.7 Infinite Loops and MAX Value Usage 30-13
 - 30.1.8 Integer Manipulation Vulnerabilities 30-13
 - 30.1.8.1 Overflow and Underflow 30-14
 - 30.1.8.2 Signed versus Unsigned Errors 30-14
 - 30.1.8.3 Truncation Errors 30-15
 - 30.1.8.4 Remedies Using Unsigned Integers 30-15
 - 30.1.8.5 Key Code Reviewing Points 30-16
 - 30.1.9 Watching Multiple Queues: Avoiding DOS Attacks FAQ 30-16
- 30.2 IOS Security Features 30-17
 - 30.2.1 AutoSecure 30-17
 - 30.2.1.1 The AutoSecure API 30-17
 - 30.2.1.2 AutoSecure Management 30-17
 - 30.2.1.3 The AutoSecure Subsystem 30-18
 - 30.2.2 How to Use Private VRFs to Isolate Interfaces 30-19
 - 30.2.2.1 Terms 30-19
 - 30.2.2.2 Guidelines for Using a Private VRF Inside the Box 30-20
 - 30.2.2.3 Example of Using a VPN to Hide the Internal Network 30-20
 - 30.2.2.3.1 Automated Configuration of the Interfaces 30-21
 - 30.2.2.3.2 NSP pmbox Interface Creation 30-23

Chapter 31 AAA 31-1

- 31.1 Overview 31-1
- 31.2 AAA Functions 31-1
 - 31.2.1 AAA Attribute Functions 31-1
 - 31.2.2 AAA Utility Functions 31-3
 - 31.2.3 Authentication and Authorization Functions 31-6
 - 31.2.4 AAA Profile Functions 31-7
 - 31.2.5 Server Group Functions 31-8
 - 31.2.6 AAA Accounting Functions 31-10
- 31.3 Authentication Call Flow 31-10
- 31.4 Authorization Call Flow 31-12
- 31.5 Accounting Call Flow 31-12
- 31.6 AAA Architecture Diagram 31-13
- 31.7 Process Interaction Diagram 31-14
- 31.8 AAA Accounting Model Using RADIUS 31-15
- 31.9 Basic AAA Configurations 31-16

31.9.1 PPP Configuration between Client and NAS Using PAP and CHAP 31-17

PART 7 Other Useful Information

Chapter 32 Scalable Process Implementation 32-1	
32.1 Introduction 32-1	
32.2 The Typical Scenario 32-1	
32.2.1 Specific Problems 32-2	
32.2.1.1 CPU Utilization 32-3	
32.2.1.2 Excessive Protocol Traffic 32-3	
32.2.1.3 Adjacency Failures 32-3	
32.2.1.4 Brittle Networks 32-3	
32.2.1.5 Random Squirrely Failures 32-3	
32.2.1.6 Pathological Process Interaction 32-3	
32.3 Addressing the Problems 32-4	
32.3.1 Process Structure 32-4	
32.3.2 Stability through Rate Control 32-5	
32.3.3 Avoiding Receive Buffer Starvation 32-6	
32.3.4 Avoiding Infinite Transmit Queues and Stale Information 32-7	
32.3.5 Complexity versus Efficiency 32-7	
32.4 Conclusion 32-8	
Chapter 33 Backup System 33-1	
33.1 Overview 33-1	
33.1.1 Operation 33-1	
33.1.2 Configuring Interfaces 33-2	
33.1.3 Specifying the Standby Interface 33-2	
33.1.4 Specifying Backup Delays 33-2	
33.1.5 Specifying Backup Loads, Main Interfaces Only 33-3	
33.1.6 Notes On Operation 33-3	
33.2 Description of Changes 33-4	
Chapter 34 Verifying Cisco IOS Modular Images 34-1	
34.1 What is a Modular Image? 34-1	
34.2 Why Create Modular Images? 34-1	
34.3 Types of Modularity Checks 34-2	
34.4 Modularity Targets 34-2	
34.5 Build Modular Images for a Single Platform 34-2	
34.5.1 Build All Modular Images for a Single Platform 34-2	
34.5.2 Build a Specific Modular Image for a Single Platform 34-3	
34.6 Build Modular Images for All Platforms 34-3	
34.7 Check Modularity with the sys/scripts/connect Script 34-4	
34.8 Modularity Checking Done by the Nightly Builds 34-4	
Chapter 35 Writing DDTs Release-note Enclosures 35-1	
35.1 What Is a Release-note Enclosure 35-1	
35.2 How Customers See Release-note Enclosures 35-1	

35.3	Who Writes Release-note Enclosures	35-2
35.4	When Do Release-note Enclosures Get Written	35-2
35.5	Writing Release-note Enclosures	35-2
35.5.1	Naming a Release-note Enclosure	35-2
35.5.2	Writing Guidelines	35-2
35.5.2.1	Conditions Under Which the Problem Occurs	35-3
35.5.2.2	Symptoms	35-3
35.5.2.3	Workaround	35-3
35.5.3	Writing Style	35-4
35.5.4	Text Formatting Guidelines	35-5
35.5.4.1	Character Formatting Guidelines	35-5
35.5.4.2	Other Formatting Guidelines	35-6
35.5.5	Guidelines for Using \$\$IGNORE in Release-note Enclosures	35-6
35.5.6	Guidelines for Using \$\$PREFCS in Release-note Enclosures	35-6
35.5.7	Sample Release-note Enclosures	35-7
35.6	Writing DDTs Headlines	35-8
35.7	Getting Help	35-8

Chapter 36 Small Feature Commit Procedure 36-1

36.1	Introduction	36-1
36.1.1	Purpose	36-1
36.1.2	Definition	36-1
36.2	Key Requirements	36-2
36.3	Commit Requirements Form for Featurettes	36-3
36.4	Featurette Commit Window	36-3
36.4.1	DDTS Required Fields and Enclosures	36-4
36.5	Featurette Mailing Alias	36-5
36.5.1	Email Format for a New Featurette	36-5
36.5.2	Email Format for a Modified Featurette	36-5
36.6	Process Overview	36-5
36.7	Documentation	36-6
36.7.1	Featurette Information From DDTs	36-6

Chapter 37 Current Cisco IOS Initiatives 37-1

37.1	Overview	37-1
37.2	Phase Containment Guidelines	37-1
37.2.1	Static Analysis Requirements	37-2
37.2.1.1	Static Analysis Resources	37-3
37.2.2	Code Review Requirements	37-3
37.2.2.1	Code Review Resources	37-4
37.2.3	Unit Test Requirements	37-4
37.2.3.1	Unit Test Resources	37-5

Appendices

Appendix A Writing Cisco IOS Code: Style Issues A-1

A.1	Purpose of This Appendix	A-1
A.1.1	Coding Conventions: Something for Everyone to Protest	A-1

A.1.2	Definitions	A-2
A.1.3	What This Appendix Addresses	A-2
A.1.4	What This Appendix Does Not Address	A-3
A.2	Design Issues	A-3
A.2.1	Do Not Abuse the Pre-Processor	A-3
A.2.1.1	Avoid Conditional Compilation	A-6
A.2.1.2	Guidelines for Using the ## Operator	A-8
A.2.2	Plan Your Feature as a Subsystem	A-8
A.2.3	Do Not Overload Existing or System Registries	A-9
A.2.4	Don't Be a Stub Slob; Use Registries	A-9
A.2.5	Don't Hog the Chip	A-9
A.2.6	Stack Size	A-10
A.2.7	Function or Macro or Inline Function?	A-10
A.2.7.1	Reasons to Use Macros	A-10
A.2.7.2	Should This Be a Macro?	A-11
A.2.7.3	Pitfalls of Macros	A-11
A.2.7.4	More About Inlines	A-14
A.2.7.5	Some Last Comments about Inlines and Macros	A-17
A.3	Using C in the Cisco IOS Source Code	A-17
A.3.1	Use ANSI C	A-17
A.3.2	Fifty Ways to Shoot Yourself in the Foot	A-17
A.4	Presentation of the Cisco IOS Source Code	A-21
A.4.1	Specific Code Formatting Issues	A-22
A.4.2	Some Comments about Comments	A-26
A.5	Variable and Storage Persistence, Scope, and Naming	A-27
A.6	Coding for Reliability	A-28
A.7	Coding for Usability	A-34
A.8	Coding for Performance	A-35
A.8.1	Performance of Algorithms and Data Structures	A-36
A.8.2	Performance Resulting from Use and Abuse of the Cisco IOS Infrastructure	A-37
A.8.3	Instruction-level Performance	A-37
A.8.3.1	Helping GCC Turn Glop into Gold	A-37
A.8.3.2	Not All Memories Are Golden	A-39
A.9	Coding for Scalability	A-40
A.9.1	Designing, Writing or Modifying IOS Code	A-40
A.9.1.1	Writing Code for a Scalable Interface Type	A-40
A.10	Coding for Security	A-42
A.11	Coding for Multiple Operating Environments	A-42
A.11.1	Definition of "Operating System Boundary" in the IOS Code Base	A-42
A.11.2	Runtime Environment and IOS Code	A-43
A.11.3	Writing IOS Code for Use with Multiple Underlying OSs	A-43
A.11.4	Issues with Using Source Code Across Multiple OSs	A-43
A.11.4.1	Dependency Management	A-43
A.11.4.2	Header File Relationships	A-44
A.11.4.3	What Is a Modularity Domain?	A-45
A.11.4.4	Implementation Incompatibilities Between OSs	A-47
A.11.4.5	Meanings and Usage of Conditional Compile Macros	A-47
A.11.5	Motivation for Creating This Set of Guidelines	A-47
A.11.6	Guidelines for Writing IOS Code That Runs Over Multiple OSs	A-48

A.11.6.1	Guidelines for Application-Code Developers	A-48
A.11.6.2	Specific Advice for Code in Source Components	A-52
A.11.6.3	Specific Advice for Code in IOS sys/tree	A-53
A.11.6.4	Guidelines for Developers Dealing with OS Ports	A-53
A.11.7	Cross OS/Platform Driver Sharing	A-53
A.11.8	Techniques for Coding	A-54
A.11.8.1	Coding for Maintainability and Safety	A-54
A.11.9	Task-Oriented Coding	A-54
A.11.9.1	Example of Application Code Usage: Choice: Remote or Local Table Access	A-55

A.12	Using Open Source Code in IOS Software	A-56
------	--	------

Appendix B Cisco IOS Software Organization B-1

B.1	12.2T Header File Location Changes for CDE	B-1
B.1.1	Files Moved to/cisco.comp/kernel/.	B-1
B.1.2	File Moved to/cisco.comp/cisco/.	B-2
B.1.3	Files Moved to/cisco.comp/ansi/.	B-2
B.1.4	Files Moved to Newly Created/cisco.comp/rc-tools/.	B-3
B.1.5	Files Moved to/cisco.comp/target-cpu/.	B-3
B.2	Description of the Cisco IOS Subsystems	B-4
B.3	Description of the IP Subsystems	B-16
B.3.1	IP Host Subsystem	B-16
B.3.2	IP Routing Subsystem	B-18
B.3.3	IP Services Subsystem	B-19
B.4	Description of the Cisco IOS Kernel Subsystems	B-20
B.4.1	Scheduler Subsystem	B-20
B.4.2	Chain Subsystem	B-20
B.4.3	Media Subsystem	B-21
B.4.4	Parser Subsystem	B-21
B.4.5	Core TTY Subsystem	B-21
B.4.6	Core Router Subsystem	B-22
B.4.7	Core Memory Management, Logging, and Print Subsystem	B-23
B.4.8	Core Time Services and Timer Subsystem	B-23
B.4.9	Core Modular Subsystem	B-23
B.4.10	Miscellaneous Subsystems	B-24
B.5	Alternatives to the Default CPU-Specific Object Directories	B-25

Appendix C CPU Profiling C-1

C.1	Overview: CPU Profiling	C-1
C.2	How CPU Profiling Works	C-1
C.2.1	Define Profile Blocks	C-1
C.2.2	Profile Block Bins	C-1
C.2.3	Tracking Ticks	C-2
C.2.4	Overhead	C-2
C.2.5	Special Modes	C-2
C.2.5.1	Notes for Dealing with CPUHOGs	C-2
C.3	Caveats about Using CPU Profiling	C-3
C.4	Use the CPU Profiler	C-3
C.5	Configure the Profiler	C-3
C.5.1	Create a Profile Block and Enable Profiling	C-4

C.5.2	Delete a Profile Block	C-4
C.5.3	Stop Profiling	C-4
C.5.4	Restart Profiling	C-4
C.5.5	Zero Profile Blocks	C-4
C.5.6	Enable Task and Interrupt Modes	C-4
C.5.7	Disable Task and Interrupt Modes	C-5
C.5.8	Enable CPUHOG Profiling	C-5
C.5.9	Display Profiling Information	C-5

C.6	Process the Profiler Output	C-5
-----	-----------------------------	-----

Appendix D Older Version of the Scheduler D-1

D.1	How a Process Stops	D-1
D.2	Queues and Process Priorities	D-1
D.2.1	Scheduler Queues	D-1
D.2.1.1	Comparison of New and Old Scheduler Queues	D-2
D.2.1.2	Compatibility Queues	D-2
D.2.1.3	Idle Queue	D-2
D.2.2	Operation of Scheduler Queues	D-2
D.2.2.1	Overall Scheduler Queue Operation	D-2
D.2.2.2	Critical-Priority Scheduler Queue Operation	D-3
D.2.2.3	High-Priority Scheduler Queue Operation	D-4
D.2.2.4	Medium- and Low-Priority Scheduler Queue Operation	D-5
D.3	Functions in the Old Scheduler	D-8
D.1	cfork() (obsolete)	D-8
D.2	edisms() (obsolete)	D-10
D.3	process_is_high_priority() (obsolete)	D-11
D.4	process_set_priority() (obsolete)	D-11
D.5	s_tohigh() (obsolete)	D-12
D.6	s_tolow() (obsolete)	D-13

Appendix E Branch Integration & Sync Processes E-1

E.1	Overview	E-1
E.1.1	Intended Audience	E-1
E.2	Terms in Branch Integration	E-2
E.3	Branch Life Cycle	E-7
E.4	Branch Modeling	E-9
E.5	Reparenting	E-10
E.5.1	Why Reparent?	E-11
E.5.1.1	Reparenting Considerations/Requirements	E-11
E.5.1.2	Reparenting Advantages/Disadvantages	E-11
E.5.2	Reparenting Scenarios	E-11
E.5.2.1	Reparent to the Grandparent Branch	E-11
E.5.2.2	Reparent to a Sibling Branch	E-11
E.5.2.3	Example of a Branch That Cannot Be Reparented	E-12
E.6	Porting	E-12
E.6.1	Why Do Porting?	E-13
E.6.1.1	Porting Considerations and Requirements	E-13

E.6.1.2	Porting Advantages/Disadvantages	E-13
E.6.2	Porting Applicability	E-13
E.7	Cloning	E-15
E.7.1	What Is Cloning?	E-16
E.7.2	Cloning Scenarios	E-16
E.7.3	Disadvantages of Cloning	E-17
E.8	Collapsing	E-18
E.8.1	Why Collapse a Branch?	E-18
E.8.2	Assumptions	E-18
E.8.3	Different Ways of Collapsing a Branch	E-19
E.8.3.1	Collapse by Converting to Task Branch	E-19
E.8.3.2	Collapse Using an Integration View	E-19
E.9	Syncing	E-20
E.9.1	Why Do Syncs?	E-21
E.9.1.1	Incremental Sync	E-21
E.9.1.2	Terminal Sync	E-21
E.9.1.3	Reparent Sync	E-21
E.9.1.4	Sync Assumptions	E-21
E.9.2	What is a Sync?	E-22
E.9.3	Elements with Child Versions	E-22
E.9.4	Bleed-through Files	E-22
E.9.5	Backing Out Some Merged Code	E-23
E.9.6	Sync Advantages/Disadvantages	E-23
E.10	Roles and Responsibilities of the Stake Holders during a Sync	E-26
E.10.1	Stake Holders Involved	E-26
E.10.1.1	Release Program Manager (PM)	E-26
E.10.1.2	Sync Engineer	E-26
E.10.1.3	Development Manager	E-27
E.10.1.4	Development Engineers	E-27
E.10.1.5	Dev Test Engineer	E-27

Appendix F Doxygen Instructions F-1

F.1	What Is Doxygen?	F-1
F.2	Why Has Doxygen Been Selected at Cisco?	F-1
F.3	What Is the run-doxygen Script?	F-2
F.4	Workspace Area Versus Archive Area	F-2
F.4.1	Workspace Area	F-3
F.4.2	Archive Area	F-3
F.5	Setting Up the Environment	F-4
F.5.1	Instructions to Set Up run-doxygen for Your Project	F-4
F.5.2	Special Notes for Remote Sites	F-4
F.6	How to Run Doxygen	F-5
F.6.1	Examples of Using the run-doxygen Script	F-6
F.7	Using the Tags	F-7
F.7.1	Escaping Special Characters	F-7
F.7.2	Doxygen Comment Blocks	F-7
F.7.3	Using Tags That Specify Descriptions	F-8
F.7.4	Descriptive Text Begins on Same Line as Tag	F-9
F.7.5	Alphabetical List of Recommended Doxygen Tags	F-9

F.8 FAQ F-11

F.9 How Can I Get Help on Doxygen? F-11

Appendix G Glossary G-1

PART 8 Index

Figures

- Figure 1-1 Cisco IOS Network Services and Protocols 1-4
Figure 2-1 Cisco IOS Fundamental Initialization Sequence 2-18
Figure 2-2 System Initialization Sequence 2-20
Figure 3-1 Operation of Scheduler Queues 3-10
Figure 3-2 Run Critical Queue 3-11
Figure 3-3 Run High Queue 3-12
Figure 3-4 Run Normal Queue 3-13
Figure 3-5 Run Low Queue 3-14
Figure 3-6 Scheduler Loop End 3-15
Figure 4-1 Regions, Memory Pools, and Chunks 4-6
Figure 4-2 Region Classes 4-7
Figure 4-3 Region Hierarchy 4-7
Figure 4-4 main and iomem Region Hierarchy 4-11
Figure 4-5 Layout of the Free-Region List after System Initialization 4-39
Figure 4-6 Layout of the Region List after System Initialization 4-39
Figure 4-7 Regions Associated with the Free-Region List and Static and Transient Memory Pools 4-39
Figure 4-8 IOS Caching Diagram 4-84
Figure 4-9 Parts of an ID from the ID Manager 4-87
Figure 4-10 Multiple views—pre-commit of new data 4-90
Figure 4-11 Multiple views—post-commit of new data 4-91
Figure 5-1 Structure of a Pool 5-7
Figure 5-2 Structure of Pool with a Cache 5-15
Figure 5-3 Packet Structure 5-18
Figure 5-4 Memory Organization within a Data Block 5-20
Figure 5-5 Comparing the pak_copy() and pak_copy_and_recenter() Functions 5-32
Figure 5-6 Input Interface Manipulation 5-36
Figure 5-7 Particle Structure 5-40

- Figure 5-8 Chain of Particles 5-41
Figure 5-9 Particle Cloning 5-49
Figure 5-10 Using Particle Clones for Separately-managed Data Areas 5-51
Figure 7-1 Developer Hooks for Platform Support 7-3
Figure 8-1 IPC Message System Interfaces 8-2
Figure 8-2 IPC Message Format- Version 0 8-7
Figure 8-3 IPC Message Format-Version 1 8-8
Figure 8-4 Non-Blocking Send for Unreliable IPC Message 8-10
Figure 8-5 Non-Blocking Send for Unreliable with Notification IPC Message 8-11
Figure 8-6 Blocking Send for Reliable IPC Message 8-12
Figure 8-7 Non-Blocking Send for Reliable IPC Message without Overlapping Sends 8-14
Figure 8-8 Non-Blocking Send for Reliable IPC Message with Overlapping Sends 8-15
Figure 8-9 Blocking Send for RPC Messages 8-16
Figure 8-10 Non-Blocking Send for RPC Messaging 8-17
Figure 8-11 IOS-IPC Process 8-18
Figure 8-12 Name Registration Message Flow 8-26
Figure 8-13 IPC Lookup Message 8-29
Figure 8-14 Open Port Message Flow for Local Port 8-29
Figure 8-15 Open Port Message for Remote Port 8-30
Figure 8-16 Close Port Message Flow 8-31
Figure 8-17 Remove IPC Port Message Flow 8-32
Figure 8-18 Current Tx IPC Message Processing through IPC Seat 8-34
Figure 8-19 Current Receive IPC Message Processing through Seat to Client 8-37
Figure 8-20 IPC Application Structure 8-51
Figure 9-1 Persistent Data Architecture 9-33
Figure 11-1 The Open Group POSIX Reference (HTML View/Download Option) 11-33
Figure 11-2 The Open Group POSIX Reference Documentation Portal Heading 11-34
Figure 11-3 System Interfaces Volume Selection from INDEX Navigation Window 11-34
Figure 11-4 Accessing the System Interfaces API Reference Index 11-35
Figure 11-5 Alphabetical List of POSIX Standard API Reference Pages 11-35
Figure 16-1 Timer States 16-3
Figure 16-2 Sample Managed Timer Tree Structure 16-17
Figure 16-3 Multiple Timer Wheel Levels Timer Model 16-21
Figure 16-4 Timer Wheel with Wheel Size = 16 16-22
Figure 16-5 Timer Wheel with Wheel Size = 256 16-23
Figure 16-6 The Elapsed Time with Different Wheel Sizes 16-23

- Figure 16-7 Single Timer Wheel Timer 16-25
Figure 16-8 Multiple Timer Wheel Timer 16-26
Figure 20-1 Flow of Event Tracer 20-29
Figure 20-2 Hashing Mechanism Components 20-63
Figure 21-1 Example of Right-Threaded Tree 21-13
Figure 22-1 Direct Queues 22-1
Figure 22-2 Indirect Queues 22-2
Figure 23-1 Essential Components of All Routers 23-7
Figure 23-2 Private Buffers 23-9
Figure 23-3 Public Buffers 23-9
Figure 23-4 Process Switched Path 23-10
Figure 23-5 Fast Switched Path 23-12
Figure 23-6 7000 Router Switch Path 23-13
Figure 23-7 RSP Memory Allocations 23-14
Figure 23-8 RSP Process Switching 23-16
Figure 23-9 RSP Fast Switching 23-17
Figure 23-10 NetFlow Switching 23-18
Figure 23-11 RSP NetFlow Switching 23-19
Figure 23-12 RSP, CEF Forwarding Tables 23-20
Figure 23-13 RSP CEF Switching 23-21
Figure 23-14 VIP Overview 23-22
Figure 23-15 dCEF Switching on Cisco VIPs 23-23
Figure 23-16 7200 Architecture 23-24
Figure 23-17 NSE 1 Architecture 23-25
Figure 23-18 NSE 1 Packet Path (1) 23-25
Figure 23-19 NSE 1 Packet Path (2) 23-26
Figure 23-20 ESR 10000 Architecture 23-27
Figure 23-21 ESR 10000 Packet Flow (1) 23-28
Figure 23-22 ESR 10000 Packet Flow (2) 23-29
Figure 23-23 GSR Architecture 23-30
Figure 23-24 FIB Tables 23-46
Figure 23-25 dCEF Switching 23-51
Figure 23-26 L2 Hardware Switching Support for an Architecture with Linecards 23-70
Figure 23-27 L2 Hardware Switching Support for an Architecture with a Hardware Engine 23-71
Figure 23-28 Hardware Session API Architecture 23-73
Figure 23-29 L2 Hardware Switching API Architecture 23-75

- Figure 23-30 Relationship Between the `12hw_switch_handle` and `12hw_switch_segment` Handles 23-75
- Figure 23-31 Position of SSM in L2 Switching Architecture 23-81
- Figure 23-32 SAL Architecture 23-82
- Figure 23-33 SAL Architecture on Distributed Platforms 23-84
- Figure 23-34 How to Obtain the Other Segment Handle 23-85
- Figure 23-35 AToM to HW API Diagram 23-102
- Figure 23-36 AC to AC HW API Diagram 23-103
- Figure 23-37 AC to L2TP HW API Diagram 23-104
- Figure 23-38 AC to VFI and PW to VFI HW Diagram 23-106
- Figure 23-39 Tunnel Switching HW Diagram 23-108
- Figure 24-1 Basic HA SSO System Platform Architecture 24-9
- Figure 24-2 Basic HA SSO System Platform Architecture with an FP 24-10
- Figure 24-3 Basic HA SSO System Software Architecture 24-13
- Figure 24-4 Basic Driver-Client Model 24-15
- Figure 24-5 Non HA Aware Protocols and Features on Switchover 24-16
- Figure 24-6 HA Aware Routing Protocols 24-18
- Figure 24-7 HA-Aware Routed Protocols 24-21
- Figure 24-8 Software Architecture of Redundancy Framework 24-31
- Figure 24-9 Client Message Routing 24-32
- Figure 24-10 Three Phases of RF Progression 24-38
- Figure 27-1 Parser Model 27-2
- Figure 27-2 Traversing the Parse Tree 27-5
- Figure 27-3 Transition Diagram for Parsing a Keyword 27-13
- Figure 27-4 Example of PRC in HA 27-38
- Figure 27-5 Zero NVGEN 27-62
- Figure 27-6 Parse Subtree of `show ip bgp <options>` 27-67
- Figure 28-1 Internet Network Management Framework Model 28-2
- Figure 28-2 MIB Object Identifiers 28-6
- Figure 29-1 Interface Manager black-box view 29-4
- Figure 29-2 IM Architecture 29-5
- Figure 29-3 Interface Stack 29-6
- Figure 29-4 Graph Representing Interface Stack 29-6
- Figure 29-5 Adding an Interface to an MLP Bundle 29-19
- Figure 31-1 AAA Architecture 31-14
- Figure 31-2 Process Interaction 31-15

- Figure 31-3 AAA Accounting Model Using RADIUS 31-16
Figure D-1 New and Old Scheduler Queues D-2
Figure D-2 Overall Scheduler Queue Operation D-3
Figure D-3 Critical-Priority Scheduler Queue Operation D-4
Figure D-4 High-Priority Scheduler Queue Operation D-5
Figure D-5 Medium- and Low-Priority Scheduler Queue Operation D-7
Figure E-1 Incremental Sync E-4
Figure E-2 The Three Contributors E-6
Figure E-3 Typical Branch Life Cycle E-7
Figure E-4 Branch Life Cycle With Porting E-8
Figure E-5 File Changed by both geo_t and georgia E-9
Figure E-6 File Changed by geo_t Branch, but Not by georgia Branch E-10
Figure E-7 File Changed by georgia Branch, but Not by geo_t Branch E-10
Figure E-8 File Changed by neither the georgia nor the geo_t Branches E-10
Figure E-9 Example of Branch Reparented to Its Grandparent Branch E-11
Figure E-10 Example of a Branch Reparented to Its Sibling Branch E-12
Figure E-11 Example of a Branch Not Reparented E-12
Figure E-12 A Port between 2 Maintenance Releases with the Same Baseline E-14
Figure E-13 Syncs Performed on Branch Branch_b E-14
Figure E-14 Changes Ported to Branch_c E-15
Figure E-15 A Sync between the 2 Port Points B and C E-15
Figure E-16 Cloning Scenario 1 E-16
Figure E-17 Cloning Scenario 2 E-17
Figure E-18 Collapsing a Branch E-18
Figure E-19 Branch Hierarchy E-22
Figure E-20 File foo.c Changes with a Sync E-24
Figure E-21 File foo.h Changes with a Sync E-25
Figure F-1 Workspace Area Versus Archive Area F-3

Tables

- Table 2-1 EHSA Functions 2-30
Table 2-2 EHSA State Meanings 2-31
Table 3-1 Cisco IOS Task States 3-17
Table 3-2 Functions for Checking and Suspending Tasks 3-21
Table 3-3 Set and Retrieve Information about a Task 3-26
Table 3-4 Process Accounting Thresholds 3-29
Table 3-5 Boolean Event Generation 3-44
Table 4-1 Common Region Classes 4-9
Table 4-2 Region Media Access Attributes 4-10
Table 4-3 Region Hierarchy Types 4-10
Table 4-4 Region Inheritance Flags 4-12
Table 4-5 Set and Retrieve Information about a Region's Attributes 4-14
Table 4-6 List of Common Memory Pool Classes 4-17
Table 4-7 malloc() Family of Functions 4-19
Table 4-8 Chunk Pool Flags 4-30
Table 4-9 Memory-Related Commands 4-43
Table 4-10 show memory Column Description 4-47
Table 4-11 show memory ecc Field Description 4-50
Table 4-12 show memory scan Field Description 4-55
Table 4-13 show process memory Column Description 4-56
Table 4-14 Mathematical Notations in VM Code 4-79
Table 5-1 pool_create() Parameters 5-8
Table 5-2 Pool flags Definitions 5-10
Table 5-3 Pool Item Vectors 5-11
Table 5-4 Dynamic Pool Parameters in pool_adjust() Function 5-13
Table 5-5 Pool Cache Item Function Vectors 5-16
Table 5-6 Guidelines for Obtaining Packet Buffers 5-24

Table 5-7	Summary of Packet Buffer Duplication Functions	5-29
Table 5-8	Input Interface Manipulation Functions for Packet Buffers	5-36
Table 5-9	Summary of Steps and Functions to Create Particle Pools	5-42
Table 5-10	Basic Functions for Particle Pool and Particle Chain Operations	5-43
Table 5-11	Summary of Particle-based Packet Data Manipulation Functions	5-44
Table 5-12	Summary of Other Particle Clone API Functions	5-50
Table 5-13	Particle Clone API Functions for Using a Specified Clone Queue	5-51
Table 5-14	show buffers Command Output Fields	5-53
Table 5-15	show buffers Pool Types	5-55
Table 5-16	show buffers Enabled Command Options	5-57
Table 5-17	show buffers Enabled Command Modifiers	5-58
Table 6-1	Subblock Function Table (sb_ft) Structure Elements	6-27
Table 6-2	Boolean use in IDBs	6-36
Table 7-1	Platform Strings	7-37
Table 7-2	Platform Values	7-40
Table 8-1	IPC Communication Services Environments	8-2
Table 8-2	Reserved Port Names	8-6
Table 9-1	nv.h Functions for Parser CLI Support	9-29
Table 9-2	nv_common.h Functions for Parser CLI Support	9-29
Table 9-3	async_chain.c Function for Parser CLI Support	9-30
Table 9-4	nv.h Platform-Specific Functions	9-30
Table 9-5	nv_common.c Platform-Specific Functions	9-30
Table 9-6	monitor1.h Platform-Specific Function	9-31
Table 9-7	nv.h Functions Called by Platform-Specific Function	9-31
Table 9-8	Platform_get_string information	9-42
Table 9-9	Memory for Persistent Variables	9-49
Table 9-10	NVRAM Performance for Both Methods	9-49
Table 10-1	Cisco IOS Socket Functions and Macros	10-1
Table 11-1	Mapping Between Unsafe Memory Functions and Safe Memory Functions	11-3
Table 11-2	Mapping Between Unsafe String Functions and Safe String Functions	11-4
Table 11-3	Available ANSI C, POSIX, and Related Library Functions	11-17
Table 13-1	Subsystem Classes in Order of Initialization	13-3
Table 14-1	Summary of Available System Registries	14-9
Table 14-2	Registry Files	14-13
Table 15-1	Functions to Get the Current Time from the System Clock	15-4
Table 15-2	Functions for Converting between Different Time Formats	15-5

Table 16-1	Macros to Determine the State of Passive Timers in the Future	16-5
Table 16-2	Functions to Determine the State of Managed Timers	16-15
Table 17-1	automore Input Prompt Processing and Results	17-3
Table 17-2	Time-String Descriptor Formats	17-4
Table 17-3	%C and %CC Descriptor Modifiers	17-4
Table 17-4	printf() Modifiers for Timestamps	17-5
Table 17-5	print() Format Codes for Timestamps	17-6
Table 17-6	Examples of Formatting Timestamps	17-7
Table 17-7	Examples of Timestamp Output	17-7
Table 17-8	printf() %a Conversion Flags	17-8
Table 17-9	Examples of Using the printf() %a Conversion Flags	17-9
Table 17-10	printf() %A Conversion Flags	17-9
Table 17-11	Examples of Using the printf() %A Conversion Flags	17-10
Table 17-12	printf() %z Conversion Flags	17-10
Table 17-13	printf() %Z Conversion Flags	17-11
Table 17-14	Examples of IPv6 Addresses	17-12
Table 18-1	Exception Signals	18-1
Table 19-1	msg_xxx.c Components	19-5
Table 19-2	msgdef() Parameters	19-8
Table 19-3	Error Message Severity Values	19-9
Table 19-4	Example of Error Messages with Good Format Values	19-11
Table 19-5	Example of Error Messages with Bad Format Values	19-11
Table 19-6	Rate-limit Macro Time Values	19-12
Table 19-7	System Message Categories	19-13
Table 19-8	Error Message Improvements	19-18
Table 20-1	Comparison of Debugging and Error Logging Facilities	20-2
Table 20-2	PROs and CONs of Event Tracing	20-3
Table 20-3	PROs and CONs of Debugging	20-7
Table 20-4	ROM Monitor Debugging Command	20-17
Table 20-5	Other Features Used by the Interface Unwedging Feature	20-23
Table 20-6	memory debug leak(s) reclaim Command Keywords	20-24
Table 20-7	EVENT_TRACE_BASIC_INSTANCE Parameters	20-32
Table 20-8	Some Defects that Address Stack Problems	20-82
Table 21-1	Functions for Searching an RB Tree	21-3
Table 21-2	Functions for Retrieving Information about an RB Tree	21-4
Table 21-3	Space Complexity of Different Trees	21-14

Table 21-4	New Threaded AVL Functions	21-15
Table 21-5	Functions for Traversing a Radix Tree	21-17
Table 22-1	Macros for Determining the State of a Queue	22-3
Table 22-2	List Action Vector Default Behavior	22-12
Table 23-1	Switching Options on High-End Routers	23-13
Table 23-2	Disabled CEF Matrix	23-31
Table 23-3	Enabled CEF Matrix	23-31
Table 23-4	dCEF Enabled Matrix	23-32
Table 23-5	FIB Subblock .h and .c Files	23-57
Table 24-1	Simplified Detection Matrix	24-34
Table 24-2	Simplified Negotiation Matrix	24-34
Table 24-3	Cisco 10000 Edge Services Router	24-59
Table 24-4	Cisco 7500 Multiprotocol Router	24-60
Table 24-5	Cisco 12000 Gigabit Switch Router	24-62
Table 27-1	Macros for Parsing Keywords	27-10
Table 27-2	Flags for Specifying Privilege Level When Parsing Keywords	27-11
Table 27-3	Flags for Specifying Other Options When Parsing Keywords	27-11
Table 27-4	Macros for Parsing Numbers	27-13
Table 27-5	New Macros for Hot ICE Compliance	27-41
Table 27-6	Data Type and Number of Stored CSB Objects	27-66
Table 27-7	Useful show parser Commands	27-73
Table 28-1	MIB Compiler Script Options	28-30
Table 28-2	MIB Compiler Script Options Supported for Backwards Compatibility	28-30
Table 28-3	MIB Repository and Workspace Directory Layout	28-52
Table 28-4	makefile Structure	28-52
Table 31-1	AAA Attribute Functions	31-1
Table 31-2	AAA Utility Functions	31-3
Table 31-3	Authentication and Authorization Functions	31-7
Table 31-4	AAA Profile Functions	31-8
Table 31-5	Server Group Functions	31-8
Table 31-6	AAA Accounting Functions	31-10
Table 35-1	Character Formatting Strings	35-5
Table B-1	Cisco IOS LAN Protocol Subsystems	B-4
Table B-2	Cisco IOS WAN Protocol Subsystems	B-7
Table B-3	Cisco IOS Bridging Subsystems	B-8
Table B-4	Cisco IOS Communications Server Subsystems	B-8

Table B-5	Cisco IOS Utilities Subsystems	B-9
Table B-6	Cisco IOS Driver Subsystems	B-9
Table B-7	Cisco IOS Network Management Subsystems	B-10
Table B-8	Cisco IOS VLAN Subsystems	B-11
Table B-9	Cisco IOS Kernel Subsystems	B-11
Table B-10	Cisco IOS IBM Subsystems	B-11
Table B-11	Cisco IOS Library Utility Subsystems	B-12
Table B-12	Cisco IOS ANSI Library Subsystems (Release 11.2 only)	B-13
Table B-13	Cisco IOS Cisco Library Subsystems (Release 11.2 only)	B-15
Table B-14	IP Host Subsystem Object Files	B-17
Table B-15	Cisco IOS IP Routing Subsystem Object Files	B-19
Table B-16	Cisco IOS IP Services Subsystem Object Files	B-19
Table B-17	Scheduler Subsystem Object Files	B-20
Table B-18	Chain Subsystem Object Files	B-20
Table B-19	Media Subsystem Object Files	B-21
Table B-20	Parser Subsystem Object Files	B-21
Table B-21	Core TTY Subsystem Object Files	B-22
Table B-22	Core Router Subsystem Object Files	B-22
Table B-23	Core Memory Management, Logging, and Print Subsystem Object Files	B-23
Table B-24	Core Time Services and Timer Subsystem Object Files	B-23
Table B-25	Core Modular Subsystem Object Files	B-24
Table B-26	Miscellaneous Core Subsystem Object Files	B-24
Table D-1	Comparison between Old and New Scheduler Functions	D-8
Table E-1	Comparison of Collapse Processes	E-20
Table F-1	Arguments for run-doxygen	F-5
Table F-2	Alphabetical List of Doxygen Tags	F-9

About This Manual

Replaced Docx with Doxygen in Document Organization. (March 2009)

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, 12.1, 12.2, 12.3, and 12.4 of the Cisco IOS software. See the documentation for the older releases under the listed release number and under “Archive” at: <http://wwwin-enged.cisco.com/doc/ios/>

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 the scalability 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:

- “Basic IOS Kernel Services,” 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 PA Plugin Driver API in Release 12.3, see *Cisco IOS Network Interface 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
 - “Interprocessor Communications (IPC) Services,” which describes Cisco IOS IPC services
 - “Standard Libraries,” which describes the standard POSIX and ANSI C library functions supported by the Cisco IOS software
 - “CNS” which describes Event Agent Services and Config Agent Services.
- 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
 - “Writing Cisco IOS Error Messages,” which describes error message services in Cisco IOS Release 11.2 and later
 - “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 paths in the Cisco IOS code and gives fast switching programming tips

- “High Availability (HA)”, which describes IOS High Availability Stateful Switchover (SSO) and Route Processor redundancy with IOS software enhancements to provide containment of software and hardware faults, and therefore an increase in network availability.
- “IP Services”, which describes new IP services supported by Cisco IOS.
- 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 New Platform,” 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
 - “MIB Infrastructure,” which explains how to support subinterfaces in the interfaces group MIB-II
 - “Security,” describes the security management features, such as the AutoSecure feature
 - “AAA” describes AAA (authentication, authorization and accounting) in Cisco IOS (New in 12.4(11)T)
- 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
 - “Writing DDTs Release-note Enclosures,” which provides guidelines for writing the text in which you report a problem in DDTs to customers
 - “Small Feature Commit Procedure,” which provides guidelines for committing very small features (“featurettes”) into Early Deployment releases
 - “Current Cisco IOS Initiatives,” which provides information on standards, procedures, and initiatives related to programming at Cisco
- Several appendixes provide the supplemental information:
 - “Writing Cisco IOS Code: Style Issues,” which documents how to write Cisco IOS 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
 - “Branch Integration & Sync Processes,” which describes the life cycle of an IOS branch.

- “Doxygen Instructions,” which describes Doxygen.
- “Glossary,” which defines some terms related to the Cisco IOS software
- Index

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.

P A R T 1

Overview

Overview

Updated the Cisco IONization 101 Guidelines link and moved the “Migrating to ION” to the end of the chapter as section 1.7. (June 2010)

Replaced Docx with Doxygen in section 1.3.7.6 “Doxygen Instructions”. (March 2009)

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

This chapter includes information on the following topics:

- Cisco IOS Software
- Network Service and Protocols
- Cisco IOS Software Components
- Scalability Changes
- Determining Infrastructure Changes in Releases
- Searching Email Alias News Archives
- Migrating to ION

1.1 Cisco IOS Software

The Cisco IOS (Internetwork Operating System) software provides a common IP fabric, functionality, and command-line interface (CLI) that is used across computer networks to support network hardware and network applications. The Cisco IOS image is loaded on routers to support network hardware devices and the network applications that use the Cisco IOS API (Application Programming Interface).

Cisco IOS runs on the processor in the router. The processor is referred to as the “Route Processor” (RP) because it does the packet forwarding or “routing”. For example, some of the general-purpose processors used for IOS-RP are:

- Motorola CPUs (the older M680x0, M68360, and the newer PPC)
- RISC (Reduced Instruction Set Computing) processors, such as IDT (Integrated Device Technologies) MIPS 4000 and 5000 series CPUs
- PMC-Sierra RM7000

When the forwarding bandwidth becomes large enough, it is cost-justified to run Cisco IOS on the Line Cards. Note that Cisco IOS has nonvirtual memory and is a non-SMP (Symmetrical Multi-Processor) operating system; that is, for each CPU in a network device there is a separate copy of Cisco IOS running.

Cisco IOS provides the unifying principles around which an internetwork can be maintained cost effectively over time. It is a software architecture, disassociated from hardware, that can be dynamically upgraded to adapt to changing technologies (hardware and software) as they evolve within a networking infrastructure. The software is written in C, except for the interrupt routines, exception routines, context switching management, and stack management that are written in assembler because they are highly processor-specific. The Cisco IOS mission to optimize the sending, receiving, and forwarding of packets between network interfaces is key to the design of Cisco IOS. The mission influences not only the design of the kernel, but also the design of various components, such as forwarding applications, network and routing protocols, and device drivers.

The original design principles used to develop Cisco IOS were based on architecture that assumed that a large network included a maximum of a few hundred interfaces and, in the beginning, IOS only supported seven interfaces per router. Cisco IOS has been revised to support new technologies, increase performance and high availability, and vastly improve scalability because the IOS framework is now required to be distributed, highly available, and scalable. For example, Cisco IOS was improved to support thirty-two thousand interfaces on c10k hardware and sixteen thousand interfaces on 7200/7400 hardware. Also, Cisco IOS has over 700 networking features. Recently, support was added to Cisco IOS for HA redundancy (a hot-failover upon either hardware or software failure).

Cisco IOS images come in over 500 “flavors”, not only for different platforms, but also for varying software within a platform, such as:

- Alternate routing protocols
- Varying network protocols
- One or more media protocols and various network interface device types
- Packet forwarding implementations, such as CEF (Cisco Express Forwarding), “fast switching”, and other options

For all of these variations, only the required code is included on the network device.

You need to keep the following in mind when working with IOS code:

- Subsystems provide customized images.

A subsystem is a discrete code module that supports various functions of an embedded system and provides an independent entry point into the system code. The key advantage to the subsystem design in Cisco IOS is to allow for customized images. You must segregate code into subsystems to meet the needs for all images. The Cisco IOS image residing in memory contains the routines to support all commands, so no disk accesses are necessary. Because of this compact implementation, only commands that are required are built in the image.

- Packets drive most processing.

Most systems, such as UNIX devices, have long-running tasks started by a command from a user. Cisco IOS is different; most processing is driven by packets that arrive on an interface. This significant operating difference affects the design of Cisco IOS.

- IOS Scatter-Gather DMA

In the past, an interface would transfer a packet directly into a single (contiguous) data buffer in memory. The ability of an interface to read directly into memory is known as DMA (Direct Memory Access) and is supported almost universally in networking interfaces. Using contiguous DMA meant that each data buffer had to be able to handle the largest packet coming in on that interface, that is, the Maximum Transmission Unit (MTU). This is a simple solution, but not very memory efficient. For example, for Ethernet packets, the average packet size is about 500 bytes, but the MTU is 1500.

A memory-efficient solution was to design a hardware interface that reads a single packet into multiple data buffers. This is known as “Scatter-Gather DMA”. The inbound interface will “scatter” the packet into multiple data buffers, known as particle data buffers; the outbound interface will “gather” those multiple buffers together for transmission of a single packet.

In looking at Cisco IOS, you will need to understand the IOS structures that are required to keep track of packets, both contiguous and particle-based, moving through the system.

1.2 Network Service and Protocols

Figure 1-1 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 internal 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.

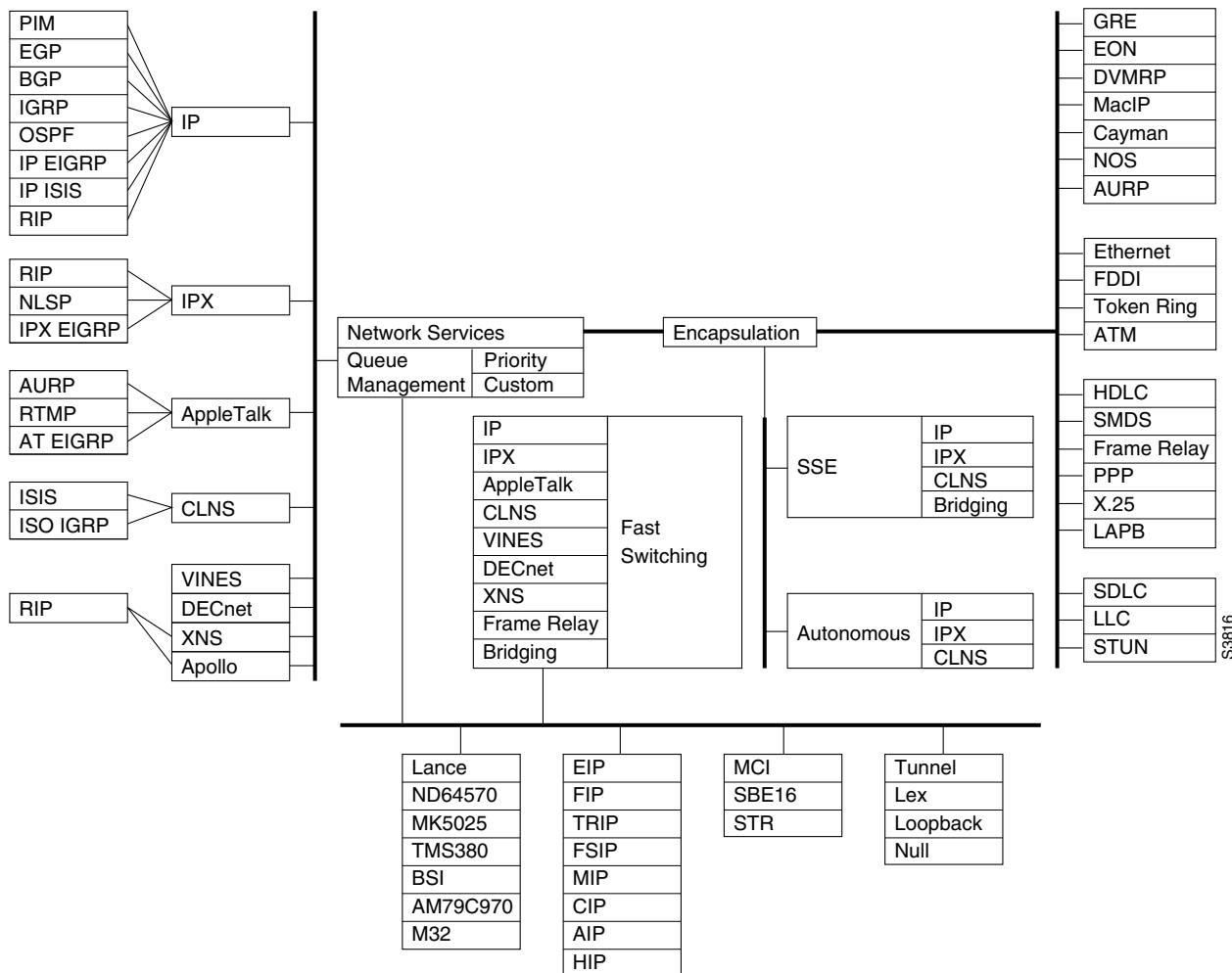
To transmit frames, the protocols use the relevant encapsulation modules—shown on the right side of the diagram—to add a media encapsulation. The encapsulation is specified by either the interface driver or via a user interface command. After the outgoing frame has been successfully encapsulated, it can be enqueued on the outgoing interface queue for final transmission. The default enqueueing 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 (Silicon Switch Engine) 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



1.3 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 Services
- Kernel Support Services
- Network Services
- Hardware-Specific Design
- Management Services
- Other Useful Information
- Appendixes

The following sections provide an overview of each component listed above.

1.3.1 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.

1.3.1.1 Basic IOS Kernel Services

The scheduler, Process Management, and Event Management comprise the Cisco IOS 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.

1.3.1.2 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.

1.3.1.3 Pools, Buffers, and Particles

Support for the handling and management of 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.

1.3.1.4 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.

1.3.1.5 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.

1.3.1.6 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.

1.3.1.7 Interprocessor Communications (IPC) Services

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

1.3.1.8 File System

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 AS5400 modems or dumping the VIP memory).

1.3.1.9 Standard Libraries

Starting with Release 11.2, the Cisco IOS software formalized its use of ANSI C library functions. The “ANSI C Library” chapter in the *Cisco IOS API Reference* described these library functions. When the POSIX functions were added, this chapter was later renamed to include all standard libraries.

1.3.1.10 CNS

Cisco Networking Services (CNS) is a set of Directory Enabled Networking (DEN) software development tools, including an implementation of DEN and DEN-derived Cisco LDAP schema, LDAP client software, and scalable event services. CNS is a controlled release, available to selected network ISV, SI, and service provider partners, as well as internal Cisco engineering teams. End-user customers will benefit from CNS via directory-enabled products that Cisco and Cisco partners bring to the market. The Cisco Networking Services (CNS) projects are delivering next-generation technologies to drive more intelligence into the network and reduce the need for human intervention.

1.3.2 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.

1.3.2.1 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 subsystem 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 specified in the makefile 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 specified in the SUBSYS_HEADER macro 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.

1.3.2.2 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.

1.3.2.3 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 three types of timers: passive timers, managed timers, and timer wheels.

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.

Timer wheels are used to provide a more efficient timer service when applications have scaling problems with managed timers. A timer wheel is a circular array of ordered timer "buckets" where the sorting has been done by the ordered array structure approach. Each bucket contains a list of timers that have the same expiration time.

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.

1.3.2.4 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.

1.3.2.5 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.

1.3.2.6 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.

1.3.3 Network Services

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

1.3.3.1 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.

1.3.3.2 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.

1.3.3.3 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 (Switch Processor), CxBus (Cisco Extended Bus), or CyBus (superset of CxBus) controller.
- Silicon switching. This class of switching is present only in routers with an SSE card.

1.3.3.4 High Availability (HA)

This chapter introduces some of the hardware, firmware, and software features that collectively provide a router's HA.

High Availability (HA) is defined as a percentage of time that the system is available for use, with a working definition of *available* as:

- The system is manageable.
- The system has discovered peer routers and has formed routing adjacencies (insofar as it is configured to do so; static routing could be used) with its peers.
- The system has negotiated and established connections with peer routers (again, as it is configured to do so).
- The system is forwarding subscriber traffic.

1.3.3.5 IP Services

The Cisco IOS software supports additional IP services:

- BEEP
- DHCP

1.3.4 Hardware-Specific Design

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

1.3.4.1 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. Portability of the code might depend on:

- Integer size.
- Data representation.
- Memory access methods of the processor.
- Assembly language function interfaces.
- Scheduling style of the operating system (preemptive or non-preemptive).

This chapter includes guidelines for writing Cisco IOS code to increase its portability using style considerations and elements designed into the code base for this purpose. It also introduces assumptions made in the bulk of the code that might influence results when porting to new platforms.

1.3.5 Management Services

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

1.3.5.1 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.

1.3.5.2 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.

1.3.5.3 MIB Infrastructure

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`
- Subinterface data structure

The `h/snmp_interface.h` file contains the subinterface 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.

1.3.5.4 Security

This chapter describes the security management features, such as the AutoSecure feature. The AutoSecure feature is managed primarily through the Cisco IOS CLI. The AutoSecure subsystem provides centralized infrastructure that can be extended for features in the future.

1.3.5.5 AAA

This chapter describes AAA (authentication, authorization and accounting) in Cisco IOS as of 12.4(11)T.

1.3.6 Other Useful Information

This section contains other useful information regarding Cisco IOS software.

1.3.6.1 Scalable Process Implementation

The Cisco IOS kernel has a lot of potential for supporting scalable, well-behaved processes that can support very large networks. Unfortunately, the track record in producing such software has been spotty. There is a lot of code that has common mistakes. The intent of this section 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.

1.3.6.2 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. 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 section is an overview of the backup system, as modified for Release 12.0.

1.3.6.3 Verifying Cisco IOS Modular Images

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.

1.3.6.4 Writing DDTs Release-note Enclosures

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.

1.3.6.5 Small Feature Commit Procedure

Cisco's IOS release processes must be sufficiently flexible to scale appropriately for the size and complexity of software commits. This document provides a streamlined approach to committing "featurettes."

Tracking features:

- Raises visibility to affected organizations such as Dev-Test, Documentation, Customer Advocacy, Marketing, and other BUs.
- Ensures that customers will be informed about the featurette via customer documentation, which is the only way customers have of getting this information.
- Allows Cisco to investigate changes to IOS to avoid lawsuits from other companies who claimed that they invented technology first.
- Alerts test teams to plan for the integration of new automated scripts to test featurettes.

1.3.6.6 Current Cisco IOS Initiatives

Cisco IOS DEs are expected to develop code according to a certain style, as well as standards, procedures, and initiatives related to programming at Cisco, such as the software development Phase Containment initiative.

1.3.7 Appendixes

The appendixes have more useful information involving the Cisco IOS software.

1.3.7.1 Writing Cisco IOS Code: Style Issues

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 WayTM 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.

1.3.7.2 Cisco IOS Software Organization

This appendix contains information on the following topics:

- 12.2T Header File Location Changes for CDE
- Description of the Cisco IOS Subsystems
- Description of the IP Subsystems
- Description of the Cisco IOS Kernel Subsystems
- What is the Purpose of the obj-4k Directory?

1.3.7.3 CPU Profiling

This appendix 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.

1.3.7.4 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 Release 11.3 of the Cisco IOS code.

This appendix 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 Cisco IOS code in releases prior to Release 11.0.

1.3.7.5 Branch Integration & Sync Processes

This appendix summarizes the life cycle of an IOS branch, beginning from its conception (branch pull) to the end of its life (collapsed). It covers various phases of the life cycle an IOS branch – typically experiences, such as incremental syncs, reparents, terminal syncs, and collapse.

1.3.7.6 Doxygen Instructions

Doxygen is an open-source tool that generates documentation for source code. It is the most widely used documentation generation tool in the industry, representing millions of lines of code.

1.4 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
- IDB Data Structure Shrinking
- IDB Subblock Modularity
- Other Scalability Changes

1.4.1 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.

1.4.2 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 PA Plugin Driver API documentation in the *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.

1.4.3 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 have affected the way that you should handle route adjustment messages.

1.4.4 IDB Data Structure Shrinking

An aspect of the scalability effort in release 12.2 is the shrinking of the core data structures so that the memory cost for a large number of interfaces is reduced. There are a number of flow-on benefits from shrinking these data structures, besides the obvious one of reducing the memory requirements:

- With smaller data structures, the caching of data within first or second level caches can be more effective. There is a greater chance of data locality, with smaller data structures.
- Migrating the fields out from a common data structure to a feature specific data structure greatly improves the software modularity of the system.
- With smaller core data structures, these data structures can be used in situations where it would have been prohibitive before, such as distributed line cards etc.
- Routers can support a larger number of interfaces without having to increase the amount of physical memory required.

The primary effort for shrinking the IDB data structures involves migrating fields from the hardware and software IDBs to subblocks. A secondary effort is to employ a more efficient scheme for storing boolean flags in the IDBs.

1.4.4.1 Dynamically Allocated Boolean Flags for IDBs

A static bit array is created in each data structure composed of an array of uints, each containing 32 bits of Boolean values, and these bits are dynamically allocated globally on a first-come, first served basis. Each request for a bit (Boolean) results in the allocation of a new index that can be used to set, clear or test the appropriate bit in the static bit arrays. Once allocated, the bit index is used for the same Boolean for all IDBs.

1.4.5 IDB Subblock Modularity

The goal for the IDB Subblock Modularity changes in release 12.2 is to allow any subsystem to allocate and install new subblocks on the IDB without having to modify generic or common code in any way, allowing new features and protocols to be added to IOS without modifying the existing IOS files.

1.4.6 Other Scalability Changes

Change	Documentation
Enhanced High System Availability (EHSA)	Chapter 2, "System Initialization"

1.5 Determining Infrastructure Changes in Releases

This section describes how to determine which infrastructure changes have been made in Cisco IOS releases. The purpose of this section is to provide guidelines to determine which changes have been made that developers need to be aware of as they write code on newer releases and when they port code back to previous releases where certain infrastructure changes may or may not be backported. Developers need to be aware of these infrastructure changes because these changes can impose different requirements on the code, depending on the release.

To determine which infrastructure changes have been made in Cisco IOS releases, you will need to use a different method depending on the type of code you are writing. For example, there is a different method for the following types of code changes:

- Major Infrastructure Changes
- Library Changes
- Feature Changes

To determine which infrastructure changes have been made in Cisco IOS releases, use one or a combination of the methods described in the next subsections, depending on the type of code you are writing. You can also check the CHANGES file, which is described at the end of this section.

1.5.1 Major Infrastructure Changes

For major infrastructure changes, go to FTS (Feature Tracking System) at:

<http://wwwin-rtim.cisco.com/fts/index.jsp>

Step 1 Under "Search Feature", click on "Keyword".

Step 2 Enter "do not market" (include the quotation marks) and click on the "GO" button.

The resulting report lists the internal changes with links to the associated documents.

1.5.2 Library Changes

For library changes, search DDTs. For example, go to the QDDTS Query Interface at:

http://wwwin-metrics.cisco.com/protected-cgi-bin/gen_query_simple.cgi

1.5.3 Feature Changes

For feature changes, search for the feature name in the geo_sync@cisco.com or flo_sync@cisco.com email archives. This way, you can find all the changes that have already been done for the feature that you want to change, per 12.2 and 12.3.

Another way to find all the feature changes per release is to use FTS. You can see the Feature IDs in each actual release under the FTS Reports tab if you click on “Pre-Packaged Reports” at:

<http://wwwin-rtim.cisco.com/fts/go?tab=report>

Or, select the “Actual Major Release” that you want and click on “Submit” at:

<http://wwwin-rtim.cisco.com/fts/go?tab=report&mode=report&page=pp&action=fbar>

1.5.4 Sample Infrastructure Changes on Newer Releases

For example, suppose that you want to add your `myfunc()` to the new P# branch that is planned to be merged into the latest technology train.

Golden List is a group of tools that will give information about the defects that occur during the life cycle of a production build. Currently, the list of defects resolved in a production build is derived through a combination of scripting and a manual process. Scripting is used to determine bugs directly committed to the branch and its parent branch. The manual process is used to remove defects backed out of syncs from the parent branch. The goal of the Golden List project is to provide:

- Tools to determine the defects resolved in a production build.
- Tools to indicate backed out and synced bugs for each sync and collapse.
- The infrastructure necessary to track the data.

See the Golden List documentation at:

<http://wwwin-enged.cisco.com/doc/ios/goldenlist/>

1.5.5 Sample Infrastructure Changes on Older Releases

For example, suppose that you want to add your `myfunc()` to the new P# branch that is planned to be merged into the old 12.0S and 12.1 train. In this case, you can check the CHANGES file in the `ccaudit` directory for information on the changes that went into each branch.

1.5.6 The CHANGES File

Look at the CHANGES file in the `ccaudit` directory for information on the changes that went into a branch (however, this is not an easy format to parse):

`/release/ccaudit/ios/branch/`

For example:

```
$ ls /release/ccaudit/ios/haw_t/CHANGE*
/release/ccaudit/ios/haw_t/CHANGES-haw_t
/release/ccaudit/ios/haw_t/CHANGES.124-3.5.T-3.6.T
/release/ccaudit/ios/haw_t/CHANGES.124-1.8.T-2.2.T
/release/ccaudit/ios/haw_t/CHANGES.124-3.6.T-3.7.T
/release/ccaudit/ios/haw_t/CHANGES.124-2.10.T-2.11.T
/release/ccaudit/ios/haw_t/CHANGES.124-3.7.T-3.9.T
/release/ccaudit/ios/haw_t/CHANGES.124-2.11.T-3.1.T
/release/ccaudit/ios/haw_t/CHANGES.124-3.9.T-3.9.T1
/release/ccaudit/ios/haw_t/CHANGES.124-2.2.T-2.9.T
/release/ccaudit/ios/haw_t/CHANGES.124-3.9.T1-4.4.T
/release/ccaudit/ios/haw_t/CHANGES.124-2.9.T-2.10.T
/release/ccaudit/ios/haw_t/CHANGES.124-4.4.T-4.7.T
/release/ccaudit/ios/haw_t/CHANGES.124-3.1.T-3.2.T
```

```
/release/ccaudit/ios/haw_t/CHANGES.124-4.7.T-4.9.T  
/release/ccaudit/ios/haw_t/CHANGES.124-3.2.T-3.3.T  
/release/ccaudit/ios/haw_t/CHANGES.124-4.9.T-5.2.T  
/release/ccaudit/ios/haw_t/CHANGES.124-3.3.T-3.5.T  
$
```

1.5.7 Related Infrastructure Initiatives

The ION, Singlesource, and possibly other groups are setting up web sites that publish the availability/support of their functionality on release trains.

ION:

<http://wwwin.cisco.com/ios/ion/releaseinfo/>

SingleSource: TBD

HA: TBD

ISSU: TBD

1.6 Searching Email Alias News Archives

To retrieve background information about issues that have been discussed previously on engineering email aliases, use the email archive search tool. Email aliases that have been archived as newsgroups can be read using a news reader in most commonly used browsers. For example, to locate the newsgroup archive for the software-d email alias, use the following steps:

Step 1 Enter the following URL in your browser window:

http://topic.cisco.com/cgi-bin/search_advanced_news_vivisimo.pl

Step 2 Select *cisco.eng.software-d* from the **Select a Newsgroup** drop-down list.

Step 3 Specify the time period through which you want to search.

Step 4 Enter the search string you want to search for then click **Search**.

Mailing lists do not have archives by default. However, it is easy to determine if someone has set up an archive.

Step 1 Go to the Mailer website:

<http://mailer.cisco.com/>

Step 2 Find the list in question using the **Display/Subscribe/Unsubscribe Mailing List** search option.

Step 3 Click the **Show Members of List** button.

Step 4 Look for a member named **News Gateway**. For example, software-d is archived and has a member named:

News Gateway : cisco.eng.software-d

Step 5 Click the browser's link to switch to the news reader option and then subscribe to the archive. This works on most browsers.

1.7 Migrating to ION

If you are working on a project that migrates Cisco IOS software to ION, you will benefit greatly by reading the *Cisco IONization 101 Guidelines* at the following URL:

PDF version:

http://wwwin-eng.cisco.com/Eng/CrossBU/CDO_Developer_Docs/ION_Developer_Docs/IONization_101_Guide/_ionization.pdf

You will find more information about ION at the following URL:

<http://wwwin-enged.cisco.com/ion/doc/>

System Initialization

This chapter includes information on the following topics:

- Overview: System Initialization
- Basic Initialization
- Cisco IOS Initialization Process
- Setting the Interrupt Levels
- Checking for Interrupt Status
- Enhanced High System Availability (EHSA)
- EHSA Overview
- EHSA Implementation Guide
- Common EHSA CLI
- EHSA Crash Handling
- IOS Warm Reboot
- IOS Warm Upgrade
- Retrieving System Information

Note Cisco IOS system initialization questions can be directed to the knox-dev@cisco.com alias.

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 EHSA Overview that points out things to consider when designing EHSA support for your platform. It continues with an EHSA 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 Cisco IOS 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.
- 3 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.
- 4 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.

Note It is interesting as a historical footnote that some platforms used a DIP switch for a config register, but it should be noted that all currently supported platforms (the ones that someone is likely to be developing for) use NVRAM or something like it.

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 this way. 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.

Note The references to older and newer need a bit of updating. We are not doing any support or development on platforms that boot this way. This is a historical footnote and doesn't apply to any currently supported platforms. For most people at Cisco, these platforms were obsolete and beyond EOE before they started working.

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 the Cisco IOS software image for parsing.
 - The user sets the config register to contain 0x02 - 0x0F. In this case, the boot configuration given by the “boot system” command will be used. If no “boot system” command is available, a broadcast TFTP attempt will be done with a hard coded image name.
- 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.
- 3 Once the image is loaded, the bootstrap terminates itself, returning control of the system to the ROM monitor.

- 4 The image is then relocated to its proper position in memory.
 - 5 The system is restarted and executes the newly loaded copy of the Cisco IOS image.
 - 6 If the image is compressed, it will self decompress and then launch the decompressed Cisco IOS image.

2.2.2.2.1 Out of Sequence Response Code

When copying an image from the TFTP boot directory to disk, you may see the following:

o stands for out-of-sequence response code, which can be due to the resulting image that is not good in the disk, or a checksum error, or TFTP operation failure.

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, such as RSP, 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.

2.2.2.4 Detailed Example of how the ROMMON Boots an Image

This is an overview of the ROMMON/bootloader boot sequence. The actual booting process of an image from ROMMON involves several steps and tracking the code may be quite confusing.

For our example, we will consider the image to be booted to be an image on the network to be loaded via “`tftp`”, one of the more complex boot sequences. `tftp` boot involves the ROMMON (ROM Monitor), the `MONLIB` (Monitor Library), and a bootstrap image.

Compressed images were not mentioned in this description because decompression is not performed by the ROMMON. Instead, the compressed image has a small routine at the beginning of it. Typically when that routine is launched, it copies its image into high memory, jumps to high memory and then decompresses that image into low memory. Finally, it launches the image in low memory.

Terms used in this section are:

- ROMMON is the Read Only Memory Monitor program. It is the program first loaded when a device is restarted or powered on. It is responsible for verifying basic hardware operation and has a minimal command line interface, for example the **boot** command, the **reset** command and the **setvar** command.
- MONLIB is the ROM Monitor Library. It has ancillary programs to support the ROM Monitor, such as device-specific **open()** and **close()** functions.

File system access on the ROMMON is separated into two layers. The lower layer, responsible for reading and writing to a certain offset on the device, is the raw access library (RALIB). The higher layer, which understands the file system on the media, is the MONLIB. The ROMMON image contains the RALIB and the MONLIB is located on the media itself. Typically, the MONLIB calls function vectors back into the ROMMON RALIB, but some MONLIBS supply RALIBs of their own.

The type of file system may change so instead of having to upgrade a ROMMON to change the file system type, an approach of placing the MONLIB on the media itself was used. The beginning of the media (usually the boot sector) contains a simple file system with one or more MONLIBS capable of reading the file system. Reading from media is a two-stage process: 1) the media MONLIB is read from the beginning of the media, 2) the loaded MONLIB is used to access the device.

There are three common file systems used by ROMMON. These are the Simple, the HES (High Ends System), and the FAT (file allocation table) file systems.

The Simple File System is a very basic file system used on media that typically don't change. The EPROM device contains this file system. The MONLIB saved on the media is also saved at the beginning of the media using this file system. The ROMMON can understand this file system natively and does not require a separate MONLIB for this file system.

The HES file system is a misnomer. It is our second generation file system. However, since it was installed on the high end platforms at the time, it has "High End" in its name. This is the proprietary system developed at Cisco for linear flash devices. It is still a relatively simple file system that must be squeezed to recover space.

The FAT file system is used as the file system on ATA disks. This is also the file system used on earlier Windows based machines.

To distinguish this file system's MONLIB from the HES MONLIB, this file system's MONLIB is typically referred to as ATAMONLIB.

- The "bootloader" image is a scaled down IOS image. It is capable of loading another image over the network. When the ROMMON does not handle a device type, TFTP: for instance, the bootloader image is used to load the networked image. Other terms in use for the bootloader image are "bootstrap", "boot image" and "boothelper". This section will use bootloader consistently.

Source files involved are:

- `/rommon/src/simpfsio.c`, the main module to manipulate files within a simple file system. This is a generic routine and there are no platform-specific variations.
- `/rommon/srcxxx/boot.c`, the main module for the boot procedure. The generic version of this file can be found in `../rommon/src/boot.c` but there are a number of platform-specific implementations found in their respective `src` directories like the `../rommon/src-4k-rsp/rsp_boot.c` file.

- `/rommon/xrcxxx/loadprog.c`, the main module to read a program into memory. The generic version of this function can be found in `../rommon/src/loadprog.c` but there are a few platform-specific implementations found in their respective `src` directories like `../rommon/src-4k-nitro/nitro_loadprog.c`.
- `/rommon/srcxxx/monfsio.c`, the main module to manipulate files within the ROMMON file system. The generic version of this function can be found in `../rommon/src/monfsio.c` and as of 12.3 there are no platform-specific variations of this routine.
- `/os/kinit.c`, the main module to start an IOS operating system, kernel initialization. The generic version of this function can be found in `../os/kinit.c` and there is only one version of this file.
- `/os/init.c`, the main module to start subsystems within IOS. The generic version of this function can be found in `../os/init.c` and there is only one version of this file.
- `os/boot.c`, the main module for reading a boot file into memory. There is only one version of this file.
- `/rommon/src/simpfsio.c`, the main module to manipulate files within a simple file system. This is a generic routine and there are no platform-specific variations.

An overview of this process described in the rest of this section is:

- 1) Both the command line boot request and `autoboot()` call the `boot()`routine.
- 2) The `boot()` routine, found in `boot.c` file, will first validate the input and machine state. Then it call the `loadprog()` function to begin the image load process.
- 3) The `loadprog()` function, found in the `loadprog.c` file, in conjunction with the `open()` function found in `monfsio.c` file, is responsible for opening and reading the image into memory. It may also (optionally) pass control to that image. Our example is for a network boot, with recursive calls to these two functions.
- 4) The `open()` function, found in the `monfsio.c` file, identifies the source which in our example is a remote network device.
- 5) The second nested invocation of the `loadprog()` function is for the bootloader program.
- 6) With the second call to the `open()` function for the bootloader, the file is on physical device (disk, slot, bootflash) on the device.
- 7) The third nested invocation of the `loadprog()` function is with for the MONLIB filename.
- 8) The third nested invocation of the `open()` function is for the MONLIB file.
- 9) Upon return to the third nested invocation of the `loadprog()` function the MONLIB program has been opened and the file descriptor returned. The entire MONLIB program needs to now be read into memory.
- 10) Upon return to the second nested invocation to the `open()` function the MONLIB program has been read into memory and can now be run. The MONLIB program will update its function vector tables with routines specific to the associated device and then open the device for the bootloader program.
- 11) Upon return to the second nested invocation of the `loadprog()` function the bootloader program has been opened and the file descriptor returned. The entire bootloader program needs to now be read into memory.
- 12) The first invocation of the `open()` function is returned to, having loaded the bootloader program.
- 13) Upon return to the first call to the `open()` function the bootloader program has been read into memory and can now be run.

- 14) The bootloader program is a stripped-down IOS image and has the same general startup as a production IOS image, with the main(), subsys_init(), start_sysinit() and scheduler() routines.
- 15) With the scheduler now running, control is passed to the init_process process to initialize subsystems and the network, among many other components. During bootloader execution it will also spawn the bootload process.
- 16) When the init_process process gives up control, the bootload process can now run. The bootload() process is responsible for loading an image from across the network. When the bootload process is complete it indicates system shutdown by setting the “system is running” boolean to FALSE.
- 17) The scheduler detects the system is no longer in running state and returns back to the main() function in the os/kinit.c file.
- 18) The main() function will notify other interested components then return to the first invocation of the open() function found in /rommon/src/monfsio.c file.
- 19) Upon regain of control the open() function will validate success of the image load.
- 20) Upon return to the first invocation of the loadprog() function the IOS image has been opened and the file descriptor returned. This execution of the loadprog() function will copy the IOS image to the correct memory location and launch it.
- 21) Unless the image is a boot image, IOS images never exit. A reload is accomplished by a soft reset in hardware.
- 22) When control is returned to the boot() function, it checks the return code, printing an error message if necessary. It then returns to ROMMON.

The following numbered lists describe the process in more detail.

- 1 Both the command line boot request and `autoboot()` call the `boot()` routine.
- 2 The `boot()` routine, found in `boot.c` file, will first validate the input and machine state. Then it call the `loadprog()` function to begin the image load process.
 - (a) Arguments are validated, setting appropriate flags. A few of the arguments are:
 - “-r” break in boot loader image
 - “-n” break in system image
 - “-a” break in both images
 - “-x” load but do not execute
 - “-v” provide verbose output during the boot procedure
 - (b) If the system is not in a clean reset state, print message and exit back to ROMMON.

```
printf("Please reset before booting\n");
return (1);
```
 - (c) If no image name or device is specified, use the default for that platform.
Example defaults are “`eprom:`”, “`flash:`”, and “`bootflash:`”.
 - (d) Parse the image name.
 - (e) Update the `bootinfo` structure with the image name and the protocol/location to be used (FTP, TFTP, SLOTO).

- (f) After successfully performing the above, the boot command then calls the `loadprog()` function to load the image. The second argument `loadonly` is established by the “`-x`” option on the `boot` command.

```
retval = loadprog(argv[optind], loadonly, argc, argv);
```

- 3 The `loadprog()` function, found in the `loadprog.c` file, in conjunction with the `open()` function found in `monfsio.c` file, is responsible for opening and reading the image into memory. It may also (optionally) pass control to that image. Our example is for a network boot, with recursive calls to these two functions.

In our example of a network boot the `loadprog()` function and the `open()` function are recursively called. The example below shows sample filenames.

```
loadprog("tftp:myid/c7200-i-mz", . . .);
open("tftp:myid/c7200-i-mz", . . .
     loadprog("%c7200-boot-mz", . . .)
     open("%c7200-boot-mz", . . .)
     loadprog("%c7200_atafslib-m", . . .)
     open("%c7200_atafslib-m", . . .)
```

The `loadprog()` function begins by calling the `open()` function specifying read only:

```
if((fd= open(loadname, O_RDONLY)) < 0) {
```

- 4 The `open()` function, found in the `monfsio.c` file, identifies the source which in our example is a remote network device.

The source is considered a network device if the device is not recognized. Thus “`tftp:`”, “`rccp:`”, and “`http:`” will each be considered a network device. In fact, on ROMMONS that don’t support ATA (Advanced Technology Attachment) standard devices (PCMCIA cards), “`disk0:`” will be considered a network device.

Standard work within the `open()` function is

- (a) Use the supplied pathname to get the associated device number.

```
if((devnum = getdevnum(path)) < 0) {
```

- (b) Use the device number to access the correct entry within the device table.

```
devptr = &devtbl[devnum];
```

- (c) Use the supplied pathname to check for a partition number.

```
partition = getdevpartition(filename);
```

- (d) Find an available file descriptor entry (`mon_fsio` structure).

```
ofptr = &openfiles[fd];
```

- (e) Fill in the file descriptor entry (`mon_fsio` structure) with the device number, file descriptor, size and partition number.

```
ofptr->devnum = devnum;
```

```
. . .
```

- (f) Execution of the `open()` function continues with one of three conditions: device specified is memory, device is a network device or it is a physical device. This is identified by the second of these tests:

```
if (devptr->flags & DEV_IS_MEMORY) {
} else if(!devptr->devsizroutine) { /* network device */
} else { /* not a netbooted image */
```

- (g) To load a network image, a bootloader is required. The bootloader filename is determined by various techniques and placed into the `shstuff` structure.
- (h) Once the bootloader is determined, `loadprog()` is called for a second time with that bootloader filename given and the `loadonly` bit (argument 2) set.

```
if(loadprog(shstuff.cmdptr, 1, 0, (char **)0) < 0 { /*load only*/
```

- 5** The second nested invocation of the `loadprog()` function is for the bootloader program.

The `loadprog()` function begins by calling the `open()` function specifying `readonly`:

```
if((fd= open(loadname, O_RDONLY)) < 0) {
```

- 6** With the second call to the `open()` function for the bootloader, the file is on physical device (disk, slot, bootflash) on the device.

- (a) The `open()` function performs the standard work identified above: finding the entry in the device table and filling in an available file descriptor table entry.
- (b) After the standard work, execution of the `open()` function continues with the condition of a physical device. This is identified by the last of these tests:

```
if (devptr->flags & DEV_IS_MEMORY) {
} else if(!devptr->devsizroutine) { /* network device */
} else { /* not a netbooted image */
```

- (c) The first step is to detect that the associated library, the `MONLIB`, is not loaded:

```
if(!(devptr->flags & DEV_LIB_LOADED || simpfile) {
```

- (d) Data from the device is read in and the magic number on the device is validated.

```
if(o_read(devnum, (char *)0,
          (char *)&magic, sizeof(magic)) <0 {
```

- (e) The next step is to read the DIB Device Information Block with `o_read()` calls.

```
if(o_read(devnum, (char *)0, (char *)dibptr,
          sizeof(struct fs_device_info_block_t)) <0 {
```

- (f) Once the DIB is read, the `MONLIB` location is determined from within it.

```
monlib_offset = dibptr->monlib_offset;
```

- (g) We check that we are working on the monitor library.

```
if(!simpfile && monlib_offset) {
```

- (h) Next the `MONLIB` filename is determined and placed into the `shstuff` Monitor Shell Stuff structure.

- A part of the DIB is a simple file system structure. This may actually contain multiple `MONLIB`s for different platforms.

- Selecting the appropriate MONLIB filename is determined by various techniques. For example, it could be from the NVRAM variable or the default name within the ROMMON image.
- The final MONLIB chosen may not even be on the current device being loaded. It could even be on the EEPROM. One possible bug here is that if the device doesn't contain a MONLIB, no MONLIB will be used. The problem here is that if that MONLIB is not for this CPU type, it won't be loadable. Since it is ELF, we probably can detect a problem and not boot the image.
- (i) With the name of the MONLIB finally chosen, `loadprog()` is called for a third time with an attempt to load the MONLIB in memory. Again the `loadonly` bit (argument 2) is set.
`if(loadprog(shstuff.cmdptr, 1, 0, (char **)0) < 0 { /*load only*/`

- 7 The third nested invocation of the `loadprog()` function is with for the MONLIB filename.

As with previous invocations, it begins by calling the `open()` function specifying readonly:

```
if((fd= open(loadname, O_RDONLY)) < 0) {
```

- 8 The third nested invocation of the `open()` function is for the MONLIB file.

There is one thing special about the MONLIB filename. It begins with **SIMPFILECHAR**, “%”. This affects the flow of processing within the `open()` function.

- (a) The `open()` function performs the standard work identified above: finding the entry in the device table and filling in an available file descriptor table entry.
- (b) As discussed previously, the `open()` function continues for a physical device, the third conditional.

```
if (devptr->flags & DEV_IS_MEMORY) {
} else if(!devptr->devsizroutine) { /* network device */
} else { /* not a netbooted image */
```

- (c) The first step is to detect that the associated library, the MONLIB, starts with a simple file character, %. and indicate this with by setting the local variable.

```
simpfile = 1;
```

- (d) The next step is to use that to determine the logic flow.

```
if(!(devptr->flags & DEV_LIB_LOADED || simpfile) {
```

- (e) As with the previous invocation, the DIB Device Information Block is read in.

- (f) The simple file system drivers, found in `/rommon/src/simpfsio.c`, are then used to open the file. The routine set in `ml_iface.open` for a simple file system is `sfs_open()`.

```
if((ofptr->fsfd = ml_iface.open(filename, mode,
                                  devnum,
                                  (int)ofptr->rfptr,
                                  ofptr->size)) < 0 {
```

The simple file system is used to read the MONLIB. This is in `rommon/src/simpfsio.c` thus for this, and thus would be `sfs_open()`.

- (g) The open MONLIB file descriptor is returned to the third calling of `loadprog()`.

```
return (fd);
```

- (h) This ends the third nested invocation of the `open()` function.

- 9** Upon return to the third nested invocation of the `loadprog()` function the `MONLIB` program has been opened and the file descriptor returned. The entire `MONLIB` program needs to now be read into memory.

- (a) Our previous execution was

```
if((fd= open(loadname, O_RDONLY)) < 0) {
```

- (b) The `loadprog()` function, upon return from the `open()` function, reads the ELF (Executable and Linkage Format) header and determines where to put the image and its starting location.

```
case ELF_MAGIC: /* ELF file magic number */
```

- (c) It then copies the image where specified within the ELF header.

```
if((numread = read(fd, (char *)loadtbl[i].paddr,
loadtbl[i].size)) != loadtbl[i].size) {
```

- (d) Finally it will set the entrypoint to be the starting point in the code.

```
entrypt = entry;
loadsize = size;
```

- (e) Because the `loadonly` indicator was set, the program is not run at this time.

- (f) The `loadprog()` function now returns to the second nested invocation of the `open()` function, indicating success.

```
return (1); /* success */
```

- (g) This ends the third nested invocation of the `loadprog()` function.

- 10** Upon return to the second nested invocation to the `open()` function the `MONLIB` program has been read into memory and can now be run. The `MONLIB` program will update its function vector tables with routines specific to the associated device and then open the device for the bootloader program.

- (a) Our previous execution was to load the `MONLIB` file.

```
if(loadprog(shstuff.cmdptr, 1, 0, (char **)0) < 0) { /*load only*/
```

- (b) Now that the `MONLIB` program is loaded in the proper place, it needs to be initialized. This is done by filling the process load block structure with the `ml_iface` structure which contains function vectors into the `MONLIB`.

```
plb.plb_addr = (char *)&ml_iface;
```

- (c) The `MONLIB` program is now run by calling the `launch()` function which is typically an assembler routine.

```
retcode = launch(PROM2_BOOTLOAD, &plb, &pib, entrypt);
```

- (d) When the `MONLIB` program runs it will

- Fill in the raw access library table with its supported vectors.

```
ralib = &ralib_vector_table;
ralib->read      = mifp->ra_read;
ralib->write     = mifp->ra_write;
```

- Fill in the monitor library interface table with its supported vectors.

```
mlifp->init      = (void *)fslib->init;
mlifp->close     = fslib->close;
mlifp->read       = fslib->read;
```

- As an example, see `filesys/rommon_fslib_iface.c`, routine `main()`.

- (e) The `MONLIB` program returns to the `launch()` function.

- (f) The `launch()` function returns to the `open()` function.

- (g) The `open()` function can now run the `MONLIB` specific open function for the bootloader device.

```
if((ofptr->fsfd = ml_iface.open(filename, mode,
                                    devnum,
                                    (int)ofptr->rfptr,
                                    ofptr->size)) < 0 {
```

- (h) The open bootloader file descriptor is returned to the third calling of `loadprog()`.

```
    return (fd);
```

- (i) This ends the second nested invocation of the `open()` function.

- 11 Upon return to the second nested invocation of the `loadprog()` function the bootloader program has been opened and the file descriptor returned. The entire bootloader program needs to now be read into memory.

- (a) Our previous execution was

```
if((fd= open(loadname, O_RDONLY)) < 0) {
```

- (b) The `loadprog()` function, upon return from the `open()` function, reads the ELF (Executable and Linkage Format) header and determines where to put the image and its starting location.

```
case ELFMAGIC: /* ELF file magic number */
```

- (c) It then copies the image where specified within the ELF header.

```
if((numread = read(fd, (char *)loadtbl[i].paddr,
                    loadtbl[i].size)) != loadtbl[i].size) {
```

- (d) Finally it will set the entrypoint to be the starting point in the code.

```
entrypt = entry;
loadsize = size;
```

- (e) Because the `loadonly` indicator was set, the program is not run at this time.

- (f) The `loadprog()` function now returns to the first invocation of the `open()` function, indicating success.

```
return (1); /* success */
```

- (g) This ends the second nested invocation of the `loadprog()` function.

12 The first invocation of the `open()` function is returned to, having loaded the bootloader program.

13 Upon return to the first call to the `open()` function the bootloader program has been read into memory and can now be run.

- (a) Our previous execution was to load the bootloader file.

```
if(loadprog(shstuff.cmdptr, 1, 0, (char **)0) < 0 { /*load only*/
```

- (b) Now that the bootloader program is loaded in the proper place, the associated process load block structure needs to be initialized.

```
plb.plb_str = bootsring;
plb.plb_addr = (char *)0;
```

- (c) Also fill in the Monitor File System I/O structure, including the offset from the start of the physical device.

```
ofptr->rfptr = (char *)((long)plb_ptr->plb_addr & phymemmask);
```

- (d) The bootloader program is now run by calling the `launch()` function which is typically an assembler routine. The first argument is set by the boot function indicating the supplied boot command options of “-a”, “-n”, or “-r”.

```
retcode = launch(boothelper_launchcode, plb_ptr, &pib, entrypt);
```

- (e) The `open()` function will gain control after the bootloader IOS image has been started, the net-booted image loaded into memory and the IOS image has shut down.

14 The bootloader program is a stripped-down IOS image and has the same general startup as a production IOS image, with the `main()`, `subsys_init()`, `start_sysinit()` and `scheduler()` routines.

- (a) The first routine to run, the `main()` function is called with an argument indicating we are loading an image.

```
boolean main (boolean loading)
{
```

- (b) The `main()` function will set the global boolean `system_loading` equal to the passed argument.

```
system_loading = (boolean) loading;
```

- (c) Next the `main()` function calls the `subsys_init()` function to initialize all subsystems, the start all subsystems of class system init.

```
subsys_init();
subsys_init_class(SUBSYS_CLASS_SYSINIT);
```

- (d) This call ends up starting a single subsystem which as part of its initialization routine will create the `init_process()` process.

```
static void start_sysinit(subsystype *subsys)
{
    pid = cfork((forkproc *) init_process, (long)system_loading,
                init_process_stack_size, "Init", startup_ttynum);
```

- (e) When control returns to `main()` it will perform other system initialization tasks such as memory setup. Eventually the system is ready for scheduled tasks and the scheduler takes control.

```
scheduler();
```

- 15 With the scheduler now running, control is passed to the `init_process` process to initialize subsystems and the network, among many other components. During `bootloader` execution it will also spawn the `bootload` process.

The important tasks of the `init_process` for this discussion are:

- (a) The `init_process()` process calls the system initialization routine to bring up major components of the `bootloader` image as well as creating the "Boot Load" process and establishing the pointer to the `plb` structure.

```
boolean system_init (boolean loading)
{
    . . .
    network_init();
    . . .
    subsys_init_class(. . .);
    . . .
    if (loading) {
        result = process_create(bootload, "Boot Load", LARGE_STACK,
                               PRIO_NORMAL);
        if (result != NO_PROCESS) {
            process_set_arg_num(result, loading);
        }
    }
    . . .
    return (TRUE);
```

- (b) The `init_process()` process then saves the arguments from the `boot` command from the process load block.

```
if (system_loading) {
    sstrncpy(buff, ((struct plb_t *)system_loading)->plb_str, MAXLEN);
}
```

- (c) Eventually the `init_process()` process gives up control allowing other processes to run.

- 16 When the `init_process` process gives up control, the `bootload` process can now run. The `bootload()` process is responsible for loading an image from across the network. When the `bootload` process is complete it indicates system shutdown by setting the "system is running" boolean to FALSE.

- (a) First the `bootload()` process will retrieve a pointer to the process load block. Remember that `plb->plb_str` contains the boot command. Note that `plb->plb_str` also is passed back to the ROMMON and contains any failure messages on exit.

```
if (!process_get_arg_ptr((void**)&plb))
    process_kill(THIS_PROCESS);
```

- (b) Next a huge buffer, the largest available space minus some operating room, is allocated to hold the downloaded image.

```
nbytes = mempool_get_largest_block(MEMPOOL_CLASS_LOCAL);
if (nbytes > MAX_BOOT_BUFFER)
    nbytes = MAX_BOOT_BUFFER;
nbytes -= (3*config_bytes);
buff = malloc(nbytes);
```

- (c) Next the `bootload()` process will locate the source of the IOS image to be loaded: the “boot” command on the console, the “boot system” commands within the image or the default image found on the device.

The image name passed to the `bootload()` process will be used in two conditions: the boot command was issued manually (configuration register set to zero) or the ROMMON autoboot explicitly requested this image by setting the `BOOTIGNORECONFIGBITS` flag.

```
if ((configregister & CFG_FILE) == 0 ||
    ((plb->plb_magic == PLBMAGIC) &&
     (plb->plb_option & BOOTIGNORECONFIGBITS))) {
    cfgbase = NULL; /* Don't try "boot" commands */

    Otherwise attempt o use the files specified within the configuration
    file.
} else {
    cfgbase = (filetype *)sysconfigQ.qhead; /* Try boot cmd list */
}
```

If you boot with any of the lower four bits in the config register set, you are autobooting. If they are zero, you get the rommon prompt. So if we get to this point and the config register is zero, someone typed in a boot request from the prompt and you should use the filename specified. Otherwise use the configured boot files.

For the second half, someone thought this. If the first item in the boot sequence calls for something that the ROMMON know to boot, there is no point in using the boot image. So instead of the first step in the autoboot sequence being to use the bootloader, attempt each local image first. If a few local images are not there, and we run into a network image, we will need to use the bootloader to boot that. The problem was if you call the bootloader, it will look at the config register, see that we are in autoboot, and start over from the beginning. That is why the `BOOTIGNORECONFIGBITS` check was added; to force the bootloader to use the given image name even if autoboot is detected.

- (d) Regardless of the source of the image filename, the next step is to read in the boot file by calling the `read_bootfile()` function.

```
if (read_bootfile(plb, pathent, buff, nbytes)) {
```

- (e) The `read_bootfile()` function will read the boot file into memory and, on a successful load, update the size and location of the downloaded image.

- Read in the file.

```
bytes_read = ifs_copy_file_to_buffer_resolved(pathent->path, bufp,
                                              size, TRUE, pathent->path);
```

- Update information in the process load block.

```
plb->plb_addr = bufp;
plb->plb_size = bytes_read;
return (TRUE);
```

- (f) Back in the `bootload()` function, information within the boot information structure and the process load block are updated with information about the booted image.

```
bootinfo->protocol = ifs_get_file_access_from_prefix(pathent->prefix);
. . .
sprintf(bootinfo->filename, "%s:%s", pathent->prefix, pathent->path);
. . .
bootinfo->ipaddr = temp.ip_addr;
```

Basic Initialization

```
    . . .
    plb->plb_str = bootinfo->filename;
```

- (g) If any messages have been printed previously, print this “end of bootloader” message:

```
%SYS-6-BOOT_MESSAGES: Messages above this line are from the
boot loader.
```

- (h) Next the `bootload()` process will set the indicator that the system is running to FALSE.

```
    system_running = FALSE;
```

- (i) Finally the `bootload()` process kills itself.

```
    process_kill(THIS_PROCESS);
```

- (j) And control is returned back to the scheduler..

- 17** The scheduler detects the system is no longer in running state and returns back to the `main()` function in the `os/kinit.c` file.

- (a) At the top of each loop through scheduling processes of each priority, the state of the system is checked:

```
while (system_isRunning()) {
```

This call equates to:

```
while (system_running) {
```

- (b) When this while loop is exited, execution falls through to exit the `scheduler()`

- 18** The `main()` function will notify other interested components then return to the first invocation of the `open()` function found in `/rommon/src/monfsio.c` file.

```
reg_invoke_hardware_shutdown();

restore_vectors();           /* Restore the system-changed vectors */
return (TRUE);
```

- 19** Upon regain of control the `open()` function will validate success of the image load.

- (a) The first step after return to the `open()` function is to check for a successful load of the IOS image.

```
if (retcode && plb_ptr->plb_size) {
```

- (b) Then the `open()` function returns to the first call of the `loadprog()` function, returning the file descriptor.

```
ofptr->mode = mode;
return (fd);
```

- 20** Upon return to the first invocation of the `loadprog()` function the IOS image has been opened and the file descriptor returned. This execution of the `loadprog()` function will copy the IOS image to the correct memory location and launch it.

- (a) Our previous execution was

```
if((fd= open(loadname, O_RDONLY)) < 0) {
```

- (b) The `loadprog()` function, upon return from the `open()` function, reads the ELF (Executable and Linkage Format) header and determines where to put the image and its starting location.

```
case ELF_MAGIC: /* ELF file magic number */
```

- (c) It then copies the image where specified within the ELF header.

```
if((numread = read(fd, (char *)loadtbl[i].paddr,
                    loadtbl[i].size)) != loadtbl[i].size) {
```

- (d) Next it will set the entrypoint to be the starting point in the code.

```
entrypt = entry;
loadsize = size;
```

- (e) Because the `loadonly` indicator was not set, control is now passed to the IOS image. This time, a NULL `plib` pointer is sent and IOS will know this is a real image and not a boohelper image.

```
if(!loadonly) {
    return (launch(image_launchcode, (struct plb *)0, &pib, entry));
```

21 Unless the image is a boot image, IOS images never exit. A reload is accomplished by a soft reset in hardware.

22 When control is returned to the `boot()` function, it checks the return code, printing an error message if necessary. It then returns to ROMMON.

```
if(retval < 0) {
    printf("boot: cannot load \"%s\"\n", bootstring);
    return (1);
}
```

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 680x0-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 680x0-based platforms use.
- 2 The generic code that follows the entry point first enables the basic GNU debugger (GDB) support for debugging.

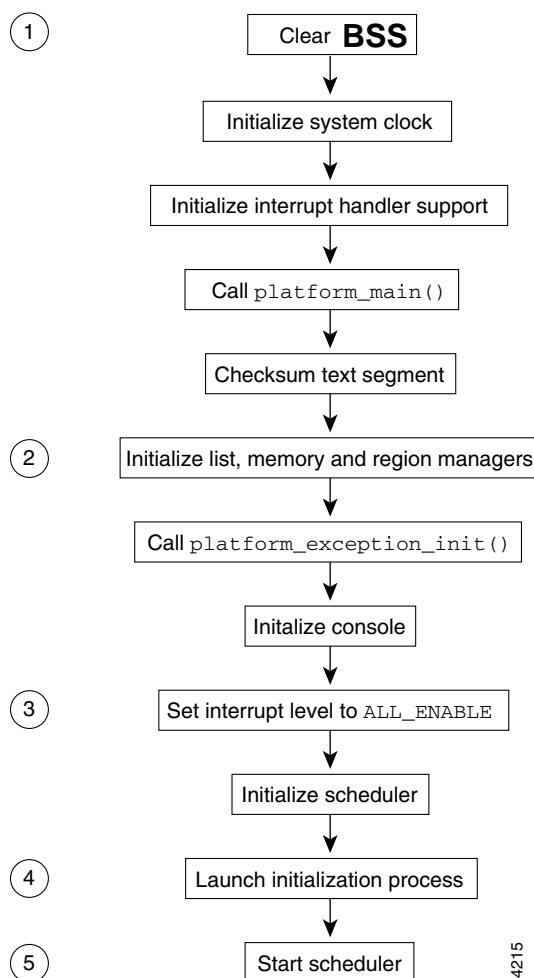
- 3 What happens next depends on where the data segment for the image is located with respect to the text segment.
- For images that are running from ROM or directly from a Flash bank, the data segment must be copied into DRAM so that it can be modified by a running system.
 - For images that are running 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 Figure 2-1 is executed before the scheduler is started and processes begin to run.

Figure 2-1 Cisco IOS Fundamental Initialization Sequence



S4215

In Figure 2-1, there are five key points during the fundamental initialization sequence:

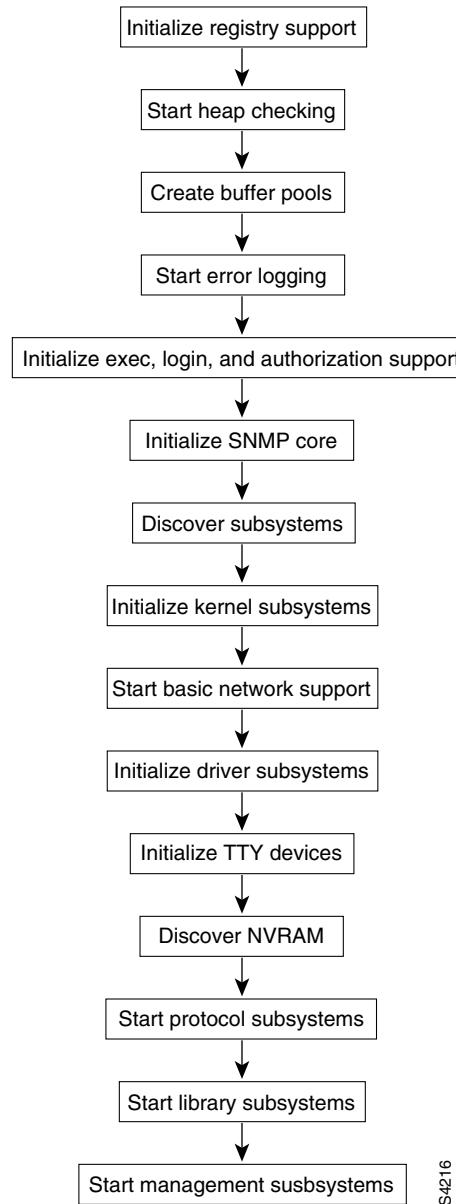
- 1 The BSS segment is cleared almost immediately at the start of the fundamental initialization sequence. (BSS is where the linker puts all global uninitialized variables.) Do not reference global data structures before this point unless you are extremely careful.

- 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 680x0-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. Figure 2-2 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



S4216

Once the initialization shown in Figure 2-2 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.

Caution One unreliable and unpopular way to manipulate the order in which commands are NVGENed is to edit your subsystem dependencies so that the subsystem whose commands you want to appear first, is declared as the dependant of the other. The reason subsystem sequencing is discouraged is that during system initialization subsystems must be ordered if sequencing is specified, so it takes longer for system initialization.

2.3.1 Parsing and Command Placement

The only order you should have is before the interfaces such as the controller command to create the interfaces, the interfaces, and after the interfaces for commands to refer to the interfaces. The order in which commands are NVGENed is dependent on the `LINK_POINT` macro and the `parser_extension_request` structure in your `*_chain.c` file. New commands should be done order independent; however, you can change the `LINK_POINT` and change the `parser_extension_request` if the order needs to be manipulated. See also the big enum in `h/parser.h` which defines `PARSE_ADD_*`.

2.4 Setting the Interrupt Levels

There are two different contexts for code to run in Cisco IOS: scheduled by the operating system (also known as the process level), versus in response to a hardware interrupt (also known as an interrupt handler or interrupt context). In most operating systems, interrupt handlers do a small amount of work, notifying code at process level that the event has occurred. Cisco IOS is unique in this respect because much more work is performed in the interrupt context, in particular the forwarding of packets is performed within the interrupt context. Interrupts other than network interrupts, such as hardware errors or console input card removal, are handled as in other operating systems, doing minimal work in the interrupt context and initiating process level work.

A packet may flow over one of the two software code paths: “fast-path” where packets are processed in the interrupt context, and process-level forwarding. Taking the fast-path, the steps in packet forwarding are all accomplished during the Rx (receive) interrupt, including FIB lookup, new frame header, and enqueue to the outbound network interface. Since this is all taking place without scheduler latency and further interrupts, it is very fast. In a normal situation, most packets will be forwarded along this path.

Process-level forwarding is used when the processing required is complex (such as when fragmentation is required) or the interrupt-context path has been disabled via the CLI. This path is longer and slower. Incoming packet traffic will preempt process-level work, and fast-path traffic can cause latencies for process-path packets. This problem is mitigated somewhat by platform-dependent network interrupt throttling (for more information, see the `throttle_netio()` function and search the code for symbols containing “`netint_throttle`”).

When an interrupt occurs it will immediately cause execution of the “interrupt handler” related to that particular type/level of interrupt. However, you can prevent immediate execution of that interrupt handler by “masking” off interrupts. When you specify a level to be masked (raised), that level and all lower level interrupts are held. When a level is unmasked (reset), that level and all higher level interrupts are released. Interrupt level values are in priority order; for example, 7: NMI (non maskable interrupts), then 6, 5, Raising and resetting the interrupt levels is used to bracket critical sections of code for protection and is necessary when working with resources which are shared between the running code and interrupt level tasks.

Overuse of this protection can cause problems. Specifically, locking for too long a time will cause interrupt level code to be delayed too long. This can cause protocol problems for latency-sensitive protocols, or flapping of adjacencies or routes. For example, 4ms is the maximum amount of time for features qualified to run while voice is running on a box. The `ISRHOG` feature was developed to support voice quality, but also exposes features that run at interrupt level for “long” periods of time.

Note When writing process-level code to protect a critical section of code, keep in mind that the code which deals with packets usually disables just `NETWORK` level interrupts. In most other places, the process-level code simply disables `ALL` interrupts.

When running in the interrupt context *or* when running at a raised interrupt level, many IOS functions are not allowed to be called. For example, `malloc()`, `free()`, `process_suspend()`, and `printf()` aren't allowed.

Note If a function is documented as unsafe to call from an interrupt routine, assume it is also unsafe to call with the interrupt level raised in the process context.

The `set_interrupt_level()` function is used to set the interrupt level initially for Cisco IOS. The following Cisco IOS functions are available to applications for raising and resetting interrupt levels:

- `get_interrupt_level()`
- `raise_interrupt_level()`
- `reset_interrupt_level()`

2.4.1 Cisco IOS Interrupt Levels

Cisco IOS supports up to 7 interrupt levels. Interrupt levels are defined on a platform basis, some in a platform-specific file name of the format `machine/cisco.xxxx.h`, and others in `cisco.comp/kernel/include/ios_interrupts_platform.h` or another file, such as `boot/diag/h/cpu_68k.h`. However, there is some consistency in interrupt levels. The highest one is NMI, the non-maskable interrupts. That supports things like attempting access to an invalid memory address and attempting execution of an invalid instruction. Usually following NMI are the levels that are required to keep the system running: hardware errors, console input, port adapter insertion and removal. Down near the bottom is Network I/O. This is because the higher levels are required to keep the box working, which takes precedence over handling a single or a few packets.

Although IOS supports 7 interrupt levels, on many MIPS platforms there is an unused interrupt level. For example, the interrupt levels for the 7200 are defined as follows:

```
#define NMI_LEVEL          7 /* NMI Timer level */
#define LEVEL_ERROR         5 /* Errors, Hot swap etc */
#define LEVEL_CONSOLE        4 /* console UART */
#define LEVEL_PA_MANAGEMENT 3 /* Port Adaptor Hi IRQ */
#define LEVEL_DMA            2 /* Galileo DMA engine */
#define LEVEL_PA_NETWORK     1 /* Network IO level */
```

2.4.2 Getting the Current Interrupt Level

To return the current interrupt level, call the `get_interrupt_level()` function.

```
#include COMP_INC(kernel, ios_interrupts.h)
int get_interrupt_level (void)
```

The interrupt level will be zero (`PLATFORM_ALL_ENABLE`) if and only if the caller is running in the process context and no interrupt levels are masked. Otherwise, it will return the level of interrupts that are masked. Using this function, the caller cannot distinguish between running in the interrupt context and running in the process context with the interrupt level raised, but that distinction generally isn't needed at runtime. When it is, the `onintstack()` function must be used.

Usually, it is only important to the calling code to find out whether it is safe to call functions that aren't interrupt-safe, i.e., whether `get_interrupt_level()` returns zero.

For example, the `get_interrupt_level()` function is used when the queuing method checks the current interrupt level in the `sys/if/fair_queue.c` file shown below. The routine containing this code is designed to be called with interrupts disabled, so it emits an error message if the design rules were not followed:

```
level = get_interrupt_level();
if (!level) {
    errormsg(&msgsym(INTVULN, LINK), level, output->hw_namestring);
}

if (!fq) {
    output->counters.output_total_drops++; /* this is the system total */
    datagram_done(pak);
    return (FALSE);
}
```

Here is another example, from `dev/flash_dvr_les.c`:

```
/*
 *   Called to wait out the long delays, eg., during VPP ON and OFF
 *   and during flash erase on sector erasable parts.
 *   Done out of kindness to others.
 */
inline void flash_delay (ulong delta)
{
    if (get_interrupt_level() == 0) {
        process_sleep_for(delta); /* delay which permits scheduling */
    } else {
        DELAY(delta); /* spin-lock: nasty but necessary */
    }
}
```

Another common use for the `get_interrupt_level()` function is within a loop, to avoid hogging the processor when in process context:

```
if (get_interrupt_level() == 0) {
    process_may_suspend();
}
```

2.4.3 Raising the Current Interrupt Level

To raise the processor level to the level specified, or keep it where it is if already at that level, call the `raise_interrupt_level()` function.

```
#include COMP_INC(kernel, ios_interrupts.h)
int raise_interrupt_level(int newlevel)
```

After raising the level, you must lower it back to the original level using the `reset_interrupt_level()` function. The code between these calls is protected from interrupts at the given level and below. This protects a resource that is shared with code that can run in the interrupt context. Changes to the resource will be atomic.

Note Both the `raise_interrupt_level()` and the matching `reset_interrupt_level()` functions should be coded in the same function to help improve the precision of detection of an unmatched `raise_interrupt_level()` during static analysis, and to improve code readability.

Setting the Interrupt Levels

For example, the `raise_interrupt_level()` function is used to raise the level to level 5, which disables the NETWORK level interrupts in the `sys/ipmulticast/pim-autorp.c` file:

```
/*
 * New Mapping - Generate SNMP notification
 */
pim_rp_mapping_change_notif(mvrf, group_prefix, mask, rp_address,
                             expire_time, PIM_NEW_RP_MAPPING);

level = raise_interrupt_level(NETS_DISABLE);
rn = rn_addroute(prm->group_prefix, prm->group_mask, mvrf->pim_autorp_cache,
                  prm->radix_nodes);
reset_interrupt_level(level);
```

Note Remember that `NETS_DISABLE` varies by platform.

2.4.4 Resetting the Current Interrupt Level

To reset the interrupt to the value returned by `set_interrupt_level()` or `raise_interrupt_level()`, call the `reset_interrupt_level()` function.

```
#include COMP_INC(kernel, ios_interrupts.h)
void reset_interrupt_level (int level);
```

For example, the `reset_interrupt_level()` function is used to reset the interrupt level after the interrupt level was raised to protect the system when changing a list for the `vfcfs_add_in_service_file()` function in the `sys/as/vfcfs_mgr.c` file:

```
/* While adding/deleting to/from the list, other tasks should not
 * be allowed to access it. They may get invalid data. This may be
 * necessary if a request to access the file list is made from
 * interrupt routines. So to avoid such events, the 'fs_state' is
 * set to 'BUSY'. The function which accesses the list checks this
 * status and succeeds only if the 'fs_state' is set to 'READY'.
 */
level = raise_interrupt_level(ALL_DISABLE);
listp->fs_state = VFCFS_BUSY;
reset_interrupt_level(level);
```

2.4.5 Setting the Current Interrupt Level (IOS only)

Warning Only the Cisco IOS should use `set_interrupt_level()`. Applications should use `raise_interrupt_level()/reset_interrupt_level()`.

To set the interrupt level to a specified value, call the `set_interrupt_level()` function.

```
#include COMP_INC(kernel, ios_interrupts.h)
int set_interrupt_level (int newlevel);
```

For example, the `set_interrupt_level()` function is used to set the interrupt level to `LEVEL_CONSOLE`, and later to set the interrupt level to `ALL_ENABLE`, for the `main()` function in the `sys/os/kinit.c` file:

```

/*
 * Install level for performing system initialization. All handlers
 * for interrupt levels above this point must be installed by now.
 */
set_interrupt_level(LEVEL_CONSOLE);

.

.

.

system_running = TRUE;           /* we are off and running */
set_interrupt_level(ALL_ENABLE); /* open up interrupts */
bootup_time = ELAPSED_TIME(time_started);

```

2.5 Checking for Interrupt Status

The `disable_interrupts()` function disables the timer without which the scheduler cannot work. Before making calls to `process_may_suspend()` to give control to the scheduler, you should check for the status of interrupts. The `process_ok_to_reschedule()` function performs that check for you. If interrupts are indeed disabled, it should raise an INTSCHED exception.

If you don't perform this check, the calling process may become suspended during which time the scheduler is unoperational, causing an eventual system crash. Because the process that caused the crash has been suspended, obtaining a correct traceback may not be possible.

2.6 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.

2.7 EHSA 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 Availability (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 implementation only, 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 externally 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 couple things: perform a boot time initialization sequence and provide the framework for higher level protocols to provide master-to-slave updates.

2.7.1 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.

2.7.2 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 Table 2-1.)

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.

2.7.3 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.

2.7.3.1 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.

2.7.3.2 Boot Parameters

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

2.7.3.3 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.

2.7.3.4 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 subsystem 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.

2.7.3.5 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.

2.7.3.6 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 slave startup-config** command.

ROM Monitor (rommon) variables and boot variables must also be auto synced.

Syncing of images is manual, and must be done by the operator.

2.7.3.7 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.

2.7.3.8 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.

2.7.3.9 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.

2.7.3.10 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.

2.7.4 Basic Flow and Operation

2.7.4.1 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\(\)](#).

2.7.4.2 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.

2.7.4.3 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.

2.8 EHSA 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*.

2.8.1 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()`.

2.8.1.1 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);
```

2.8.2 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
<code>ehsa_event()</code>	Tells EHSA to switch state.
<code>ehsa_get_state()</code>	Gets the current EHSA state.
<code>ehsa_suspend_init()</code>	Suspends the init process.
<code>ehsa_resume_init()</code>	Causes the init process to resume.
<code>ehsa_secondary()</code>	Become an EHSA Secondary: suspends the init process and starts the EHSA Secondary background process.
<code>ehsa_primary()</code>	Starts the Primary background process.
<code>ehsa_init_control()</code>	Initialize the control structure, <code>ehsa_control_t</code> .
Registry Call	Description
<code>ehsa_switch_to_primary(void)</code>	A list registry for functions that need to be run when a Secondary transitions to Primary. Invoked from <code>ehsa_event()</code> .

Table 2-1 EHSA Functions (continued)

API Call	Description
<code>ehsa_switch_to_secondary(void)</code>	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 <code>ehsa_event()</code> .
<code>ehsa_switch_to_standalone(ehsa_event_t old_state)</code>	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 <code>ehsa_event()</code> .

2.8.2.1 Actions on Status-State Transitions

Table 2-2 describes the EHSA state meanings.

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 <code>main()</code>). This is also the state that should be set when the hardware detects that the other EHSA card has been pulled from the chassis.	<code>ehsa_event(EHSA_SW_SA)</code>	<code>reg_invoke_ehsa_switch_to_standalone()</code> 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 <code>ehsa_secondary()</code> 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.	<code>ehsa_event(EHSA_SW_P)</code>	None. See <code>ehsa_primary()</code> in the API section above.	EHSA_STANDALONE, EHSA_DEAD
EHSA_SECONDARY Two EHSA-capable cards exist and this one is Secondary.	<code>ehsa_event(EHSA_SW_SD)</code>	None. See <code>ehsa_secondary()</code> in the API section above.	
EHSA_DEAD This CPU is dead or dying. Last EHSA state before returning to ROM Monitor (rommon).	<code>ehsa_event(EHSA_SW_D)</code>	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.	

2.0.0.1 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_PRIMARY`.
- 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 {
            ttypprintf(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(CTTY, "\nnehsa_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_SECONDARY`.

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_PRIMARY`.

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.c Fri Apr 17 13:06:30 1998
--- new_init.c Fri 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 development.

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 a 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 *keywords* depends on the *object-type*.

2.1.2.2 Redundancy Display

```
> 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 any 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 EHSA 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 EHSAs 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 EHSAs 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 EHSAs 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 flush 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.

2.3 IOS Warm Reboot

This section includes a description of the IOS Warm Reboot facility and instructions on using the IOS Warm Reboot API. For more detail on the design of IOS Warm Reboot, see the “*IOS Warm Reboot Capability Software Unit Functional/Design Specification*,” [EDCS-238608](#).

IOS Warm Reboot was developed to cut IOS reboot time. IOS Warm Reboot is available in 12.3T, 12.2S-RLS3_ITD1, and is planned for 12.2S. The gains achieved with IOS Warm Reboot are made by eliminating the need to access flash, read flash, and decompress the image.

IOS Warm Reboot allows IOS to go back to the start of the text segment in memory and restart execution from it. A copy of the initialized variables is kept in memory and is used to overwrite the existing location of memory where the initialized variables are stored for use by the executable. The major gains in warm rebooting can be attributed to the fact that nothing will need to be copied from flash to the RAM on a *warm-reboot*. The text and read-only data segments remain in memory, the read-write data segment is restored, and the BSS and heap are rebuilt from scratch. Thus, when the CPU jumps back to the start of the text segment and starts operating, it will be no different (except in time) than if the binary had been read from flash, decompressed, and execution begun.

IOS Warm Reboot can be enabled only on platforms that support ECC (Error Checking and Correcting), such as the C3700, C7200 / NPE-400, NPE-1G, RSP4+, RSP8, C10000, C12000, C6500/C7600, and so on. Release notes should point this out and also point out the additional memory consumption of IOS Warm Reboot.

2.3.0.1 Main Benefits of IOS Warm Reboot

IOS Warm Reboot provides the following main benefits:

- 1 A savings of a few seconds to a few minutes in bootup time, depending on the platform. The savings will only be in flash access, flash read, and image decompression. The config parsing will still need to be done. On a dual-boot platform (where a `bootldr` image loads, and then a main image), the savings will be greater because the need to boot the `bootldr` and parse the configuration file after loading the `bootldr` image is eliminated.
- 2 If a flash card is removed, a router is not rendered useless because it can still reboot, provided it is not forced into a cold boot (for example, due to a power failure).
- 3 Allows more aggressive compression algorithms on the C2600/C3600 platforms because the cost of decompression is only paid for a cold boot (warm reboots do not require a decompression).

This IOS Warm Reboot documentation includes the following subsections:

- List of IOS Warm Reboot Terms
- List of IOS Warm Reboot Functions
- IOS Warm Reboot CLI
- IOS Warm Reboot Flow Overview
- IOS Warm Reboot Detailed System Flow
- Future IOS Warm Reboot Implementation
- Data Structure Support for IOS Warm Reboot
- How to Make a Platform IOS Warm-Reboot-Aware

2.3.1 List of IOS Warm Reboot Terms

The following words, acronyms, and actions are used throughout this discussion about IOS Warm Reboot and may not be readily understood:

cold reboot

An IOS reload wherein the ROM Monitor (ROMMON) copies the configured image from a storage device (like flash memory) into main memory, the image is decompressed, and execution is started.

warm reboot

An IOS reload without ROMMON intervention, wherein the image restores the read-write data from a previously saved copy in the RAM, and starts execution. This does not involve a flash-to-RAM copy and image self-decompression.

warm reboot storage

The RAM area used to store the data segment and bookkeeping information required for the warm reboot feature to work.

2.3.2 List of IOS Warm Reboot Functions

The following API functions are required for IOS Warm Reboot (see the reference pages in the *Cisco IOS API Reference* for detailed information on these functions):

- 1 [reg_add_wrb_handle_return_to_rommon\(\)](#)
- 2 [reg_invoke_wrb_handle_exception\(\)](#)
- 3 [wrb_get_storage_end\(\)](#)
- 4 [wrb_handle_exception\(\)](#)
- 5 [wrb_handle_return_to_rommon\(\)](#)
- 6 [wrb_is_wrb_signal\(\)](#)
- 7 [wrb_platform_attempt_warm_reboot\(\)](#) Must be written by the developer.
- 8 [wrb_platform_disallow_config\(\)](#) Must be written by the developer.
- 9 [wrb_platform_post_data_restore_init\(\)](#) Must be written by the developer.
- 10 [wrb_platform_pre_data_restore_init\(\)](#) Must be written by the developer.
- 11 [wrb_platform_reset\(\)](#) Must be written by the developer.
- 12 [wrb_platform_return_warm_reboot_storage\(\)](#) Must be written by the developer.
- 13 [wrb_restore_data_segment\(\)](#)
- 14 [wrb_save_data_segment\(\)](#)
- 15 [wrb_will_reload_warm\(\)](#)

2.3.3 IOS Warm Reboot CLI

The following CLI commands constitute the end user's interface for IOS Warm Reboot:

- **warm-reboot**
- **reload**
- **show warm-reboot**

2.3.3.1 Warm-reboot config Command

Note The **warm-reboot** configuration is disabled by default.

The warm reboot feature does not rely on the parser-configuration phase to indicate whether the data segment needs to be saved or not, because this phase is quite late in the bootup sequence and the data segment would have changed by then. Instead, a ROMMON variable is used to indicate whether the data segment needs to be saved or not. If you want to enable **warm-reboot**, you need to set the CLI (with the **warm-reboot** command) and initiate a cold-reboot, so that the data segment can be saved the next time IOS starts up.

When **warm-reboot** is enabled, the following warning message is generated to explain that warm reboot will not become active until the next cold reboot (because of the need to make a copy of the initialized memory section):

```
warm-reboot will not be active until the next cold reboot has been completed.
```

When **warm-reboot** is disabled, then the memory consumed by the **warm-reboot** storage is given back to the heap.

Here is the **warm-reboot** command help:

```
router(config)#warm-reboot ?
  count   Set max number of warm reboots in a row
  uptime  Set the uptime after which warm reboot is safe in case of a crash
<cr>

router(config)#warm-reboot count <1-50> uptime <0-120>
```

The **count** is the user-configurable maximum number of warm-reboots allowed between any intervening cold-reboot. The default value is five.

The **uptime** is the user-configurable time in minutes that must elapse between system configuration (from startup configuration) and an exception, for a warm reboot to be attempted. If the system crashes before this time elapses or before the system has been configured, **warm-reboot** will not be attempted and the system will fall back to cold-reboot. The default value is five minutes.

2.3.3.2 Reload exec Command

Even when you have enabled **warm-reboot**, executing a **reload** command will result in a cold reboot by default. This allows users who have forgotten that they have enabled **warm-reboot** and who reboot their router after changing the IOS image to not be surprised. Instead, a new addition (shown below) was added to the reboot CLI to support IOS Warm Reboot when a **reload** command is issued.

```
router#reload ?
  LINE      Reason for reload
  at       Reload at a specific time/date
  cancel   Cancel pending reload
  in       Reload after a time interval
  warm    Reload should be warm
<cr>

router#reload warm ?
  LINE      Reason for reload
  at       Reload at a specific time/date
```

```

cancel      Cancel pending reload
in         Reload after a time interval
<cr>

```

Note Warm reboot takes place in any of the above exec CLI cases only if it had been configured before the previous cold reboot.

The **config-register** command is used to control system bootup. However, if you have enabled **warm-reboot**, the settings in the **config-register** will be ignored and the system will initiate a warm reboot.

2.3.3.3 Show warm-reboot Command

```

ios104#show warm-reboot
Warm Reboot is enabled

Statistics:
X warm reboots have taken place since the last cold reboot
xxx KB taken up by warm reboot storage

```

2.3.4 IOS Warm Reboot Flow Overview

This section describes a typical implementation and execution of IOS Warm Reboot, followed by information on the IOS Warm Reboot API functions required for warm reboot.

The following steps detail a typical implementation and execution of IOS Warm Reboot.

Step 1 Load an image containing the IOS Warm Reboot code. This additional code, discussed in detail within this section, covers:

- Platform-independent code
- Processor-dependent code—The MIPS 4K processor code is provided as a guideline.
- Platform-dependent code—The C7200 is provided as a guideline.

Step 2 Enable warm reboot with the **warm-reboot** enable configuration command:

```
ios(config)# warm-reboot
```

To display the status of warm reboot, use the privileged **show** command:

```
ios# show warm-reboot
```

Step 3 Save the running configuration to NVRAM.

```
ios# write mem
```

Step 4 Perform a cold reboot to establish the structures required for a later warm reboot:

```
ios# reload
```

This cold reboot will perform the additional tasks discussed below. They are covered in more detail later in this chapter.

- Establish the Warm Reboot save area. For the 7200, this is carved from the beginning of the heap by platform-dependent code.

- Compress and save the Warm Reboot header and the text and data areas in the save area established above.

Step 5 Perform a warm reboot (a) using the CLI or (b) due to an exception.

- (a) Issuing the privileged **reload warm** command performs a warm reboot:

```
ios# reload warm
```

When the user requests warm reboot, a warm reboot is executed even if it is immediately after a reload; the system does not check if 5 minutes have elapsed since the system configured itself from NVRAM and the system does not check if there has been an exception.

- (b) Exceptions (SIGILL, SIGBUS, SIGSEGV, SIGWDOG, and SIGRELOAD) will automatically perform a warm reboot, restricted by the following:

- If IOS crashes within 5 minutes after the system has been configured, it will fall back to cold-reboot. The user is provided a CLI to override the default five-minute time allowed.
- If IOS crashes even before system has been configured, then it will cold-reboot.
- If IOS has warm rebooted five times consecutively, then it will perform no further warm reboots. This limit is also configurable and there is no time limit here; for example, the five consecutive warm reboots may well have happened over a period of a month.
- Another restriction is that the text and the saved data are checked for corruption and a warm reboot is attempted only if they are found to be OK.

2.3.4.1 Warm Reboot Platform-Independent Functions

This code is used to carry out platform-independent tasks. These tasks are:

- 1 At cold-startup, detect whether `warm-reboot` has been enabled by the user.
- 2 Compress and save the data segment in the `warm-reboot` storage.
- 3 Save the `warm-reboot` bookkeeping state.

For example, the following functions are expected to be called for the 4k platforms in `main_4k.c` to save and restore the data segments:

- 1 `wrb_get_storage_end()`
- 2 `wrb_is_wrb_signal()`
- 3 `wrb_restore_data_segment()`
- 4 `wrb_save_data_segment()`
- 5 `wrb_will_reload_warm()`

2.3.4.2 Warm Reboot Platform-Independent Registry Services

The `reg_invoke_wrb_handle_exception()` registry service is expected to be called from processor-specific code only; for example, from `handle_exception()` in `gdb_4k.c`:

```
reg_invoke_wrb_handle_exception(sig, code, sc);
```

2.3.4.3 Warm Reboot Platform-Specific Functions to Write

IOS Warm Reboot requires support from platforms. These functions (hooks) are specific to each platform for supporting `warm-reboot` and must be written by you, the developer, for each platform.

The following tasks will need to be carried out by your platform-specific code:

- 1 Change the linker script to mark the beginning of RW data. Indicate the start and end of the warm-reboot storage. Each platform is free to choose any location for reserving the required space. See section “How to Make a Platform IOS Warm-Reboot-Aware,” for an example of the change to the linker script.
- 2 Call the `wrb_get_storage_end()` function to find the end of the saved data in `platform_memory_init()`, and to create a region that contains this memory area. In the same function, make any other adjustments based on the end of saved data. For example, you had decided to place warm reboot storage at the end of BSS, you would need to adjust the start of the heap segment.
- 3 Return the warm reboot storage region created in step 2 to the processor mempool when warm reboot is disabled.
- 4 Carry out any platform-specific reinitialization before the data segment is restored. This includes disabling interrupts, watchdogs, and resetting port adapters, etc.

Keep in mind:

- The code added for `warm-reboot` is functionally split in such a way that unnecessary code is not executed for platforms not supporting `warm-reboot`. The code is organized into platform-independent, processor-dependent, and platform-dependent parts.
- The integrity of the saved data segment should be verified before `warm-reboot` is initiated. Also, any bookkeeping state that the `warm-reboot` feature maintains should be check-summed and verified before use. On platforms supporting MMU, the saved data segment and any other state maintained by `warm-reboot` should be protected as read-only. The integrity of segments that persist in memory (across warm-reboots) should also be verified before initiating `warm-reboot`. These segments include the text and the read-only segment. Any failure in these integrity checks should cause the device to fall back to cold-reboot.
- Platforms may need to identify and add any additional sanity checks that need to be carried out before initiating `warm-reboot`. To make this convenient, the `warm-reboot` feature provides “hooks” so that the platforms can carry out any necessary reinitialization. These hooks are provided by the IOS Warm Reboot API functions.

The following API functions must be written for each platform in order to install IOS Warm Reboot:

- 1 `wrb_platform_attempt_warm_reboot()`
- 2 `wrb_platform_post_data_restore_init()`
- 3 `wrb_platform_pre_data_restore_init()`
- 4 `wrb_platform_reset()`
- 5 `wrb_platform_return_warm_reboot_storage()`

In addition, your platform might require modification to the following registry services because some of the routines provided by these services might assume that we are returning to ROMMON, which would not be true in the case of warm reboot. So you might need to separate out the actions to be taken before warm rebooting and have them be done before going into ROMMON.

- `reg_add_system_exception` routines
- `reg_add_hardware_shutdown` routines

Registry services like `system_exception` and `hardware_shutdown` allow platforms to do cleanups before going down. If any of the routines added to these services expect ROMMON to do something, that thing will not happen in the case of a warm reboot since ROMMON is not used by IOS Warm Reboot. So, it should be handled by your platform. Warm reboot code sets a flag before

invocation of these services, which will be available to these added routines via the service. This flag indicates whether the platform will warm reboot. Based on the value of this flag, assigned tasks from these routines can be skipped. In case of a warm reboot failure, these skipped tasks need to be carried out before falling back to ROMMON. For this purpose another registry service will be provided by warm reboot, which will be invoked in case of a warm reboot failure.

The following new registry service might also need to be implemented, depending on what happens for your platform before going into ROMMON:

- `reg_add_wrb_handle_return_to_rommon`

For example, the 3660 platform adds the `c3660_handle_return_to_rommon()` function to the `system_exception` and `hardware_shutdown` registry services. These are called when the system is returning to ROMMON due to an exception (`system_exception`) or due to the **reload** CLI (`hardware_shutdown`). In the `c3660_handle_return_to_rommon()` function, besides other things, the platform tells the ROMMON to reset gt64120. It turns out that this does not make sense for warm reboot because ROMMON is not involved in a warm reboot. This is something very platform-specific and should not be carried out for warm reboot. In fact, this should be carried out only in the case of a warm reboot failure. Platform developers are expected to find out such platform-specific things and take care of them. IOS Warm Reboot provides a facility to take care of such things.

For the specific case above, it has been taken care of as below: The statement in the `c3660_handle_return_to_rommon()` function that “tells the ROMMON to reset gt64120” is to be protected by an `if` construct like the following:

```
if (!wrb_will_reload_warm()) {  
    emt_call(EMT_SET_RESET_TYPE); <= this is what "tells the rommon to  
    reset gt64120"  
}
```

Now, this works well because we don't do this when we are about to warm reboot. The problem, eventually, comes when warm reboot fails and we have to fall back to ROMMON, because we would have done so without “telling the ROMMON to reset gt64120.” It is to take care of this situation that we have to add a new function to the `wrb_handle_return_to_rommon` warm reboot registry service. So we put the skipped task(s) in a new function, as shown here:

```
void c3660_set_reset_type (void)  
{  
    emt_call(EMT_SET_RESET_TYPE);  
}
```

and add this function to `wrb_handle_return_to_rommon()`. If warm reboot fails now, it will call this registry service before falling back to ROMMON and we would not skip “telling the ROMMON to reset gt64120” before falling to ROMMON. The platform developers are expected to investigate such tasks specific to their platforms when making them warm-reboot-aware.

In addition, platform developers can implement the `wrb_platform_disallow_config()` function and do a `reg_add_wrb_platform_disallow_config()` if they want to conditionally enable a warm-reboot config. This is optional (only interested platforms can do so). The `wrb_platform_disallow_config()` function can check whether there is enough DRAM available for warm-reboot and/or any other conditions that should be satisfied before enabling warm-reboot. If enough memory is not available, it should print an error message indicating the reason to disallow warm-reboot config and return TRUE. Otherwise, if everything is fine for warm-reboot config, it should return FALSE.

2.3.4.4 Processor-Specific Warm Reboot Support

Warm reboot support is common to all platforms for a given processor. The following tasks are processor-specific:

- 1 On cold-startup, save the register context and the text-entry point. This is done by calling `set jmp()` and is the first thing that is done in the IOS entry function (`__start`), in `main_4k.c`.
- 2 The exception-handling sequence needs to make an IOS Warm Reboot API call before returning to ROMMON, so that warm-reboot can be initiated if required. For example, R4k platforms have this call in the `handle_exception()` function in `gdb_4k.c`.
- 3 The exception-handling sequence needs to set a flag that denotes that a warm reboot is about to take place, so that platform code added to the `system_exception` service can skip any tasks that assume ROMMON intervention on reload or can handle them in IOS.
- 4 In case of a warm reboot failure, the exception-handling sequence needs to call a warm reboot service to which the platforms can add any tasks that were skipped due to 3 above.

2.3.5 IOS Warm Reboot Detailed System Flow

The IOS Warm Reboot system flow can be described in four logically separate flows:

- Enabling/Disabling IOS Warm Reboot
- Warm-Reboot-Aware System Startup
- Warm Reboot Exception Handling
- User-Initiated Warm Reboot

2.3.5.1 Enabling/Disabling IOS Warm Reboot

Enabling warm reboot via the **warm-reboot** configuration command does the following:

- 1 If `warm-reboot` is being enabled for the first time, a message is displayed indicating that IOS will be capable of warm-rebooting only after the next cold-reboot.
- 2 If the user has specified the count of maximum warm-reboots (via the **count** argument that can be 1 to 50), it sets that value; otherwise, the default value of 5 is used.
- 3 If the user has specified the minimum uptime for which a warm reboot will be attempted (via the **uptime** argument that can be 0 to 120), it sets that value; otherwise, the default value of 5 minutes is used.
- 4 When the configuration is saved to NVRAM after enabling this CLI, the ROMMON `WARM_REBOOT` variable is set to `TRUE`. This allows any subsequent image loading (cold rebooting) to initiate the system startup warm reboot sequence. This variable can also be set at the ROMMON prompt to get the same effect.

Disabling warm reboot via the **no warm-reboot** configuration command does the following:

- 1 The ROMMON `WARM_REBOOT` variable is set to `FALSE`.
- 2 The `wrb_platform_return_warm_reboot_storage()` platform-specific function is called to return the warm reboot storage to the heap.

2.3.5.2 Warm-Reboot-Aware System Startup

System startup is the path of execution taken after control is transferred to the IOS text entry (`__start` in `main_4k.c` for R4k systems). The tasks that must be carried out on a system startup are logically separated out for you into the following subtasks:

- Save the Startup Context
- Save the Data Segment
- Initialize the Platform Memory

2.3.5.2.1 Save the Startup Context

As early as possible in the startup function, IOS needs to save the register context, which must be restored while warm-rebooting. This is processor/platform-specific and is done by calling `setjmp()`.

2.3.5.2.2 Save the Data Segment

The data segment must be saved before any initialized global/static variables are changed. For most platforms, this will be done inside the IOS startup function at a suitable point. A call will be made to the `wrb_save_data_segment()` function, which will carry out the following steps:

- Step 1** Read the ROMMON variable that indicates whether `warm-reboot` has been enabled or not. If `warm-reboot` has not been enabled, the remaining steps in this flow do not apply and the standard reload is performed.
- Step 2** Get the `warm-reboot` storage location identified in the linker script. (See the variables `_wrb_start_storage` and `_wrb_end_storage` in section “How to Make a Platform IOS Warm-Reboot-Aware”).
- Step 3** Reserve space in the `warm-reboot` storage for the `warm-reboot` header. The `warm-reboot` storage will be used to store not only the saved data segment, but also a `warm-reboot` header (`wrb_saved_hdr_t`).
- Step 4** Compress and save the data segment after the `warm-reboot` header. If there is not enough space available for saving the data segment, compression will fail.
- Step 5** Compute the end of the compressed data segment, and save this value in the `warm-reboot` header.
- Step 6** Calculate checksums for text and read-only sections and save them in the `warm-reboot` header.
- Step 7** Calculate checksums for the saved data and `warm-reboot` header, and save them in the `warm-reboot` header.

Failure to carry out any of these steps will result in the warm reboot data not being stored and the system will resort to a cold reboot when the next crash occurs.

2.3.5.2.3 Initialize the Platform Memory

Platform-specific memory initialization is done inside the `platform_memory_init()` function. This function carves out the regions and the memory pools for that platform. The `platform_memory_init()` function performs the following steps on a warm-reboot-enabled system:

- Step 1** Calls `wrb_get_storage_end()` to get the end of the `warm-reboot` storage.

- Step 2** Creates a region to encompass memory from `_wrb_start_storage` to the end of the `warm-reboot` storage defined above. By convention, this is called the “`main:saved-data`” region. Sets a flag to remember that this region was created, because later, if warm reboot is disabled, we will need to return this region to heap.
- Step 3** Depending upon where `warm-reboot` storage has been placed by the platform, the start of the heap may have to be adjusted. If the `warm-reboot` storage is placed after BSS, the platform needs to adjust the start of its heap segment accordingly (so as to not overwrite the `warm-reboot` storage).

2.3.5.3 Warm Reboot Exception Handling

Here we describe the additional tasks that are carried out to conduct a `warm-reboot` in the case of an exception. The changes are made to the common exception handler function (`handle_exception`, in `gdb_(cpu).c`) files.

The existing exception handling sequence continues to remain the same; it is only at the end of the exception handler that, instead of returning to ROMMON, a `warm-reboot`-specific function is called to attempt a `warm-reboot`. The IOS Warm Reboot feature avoids changes to ROMMON. This means that if necessary, IOS may carry out operations that were originally carried out by the platform’s ROMMON before rebooting an IOS image.

2.3.5.3.1 Sample Standard Exception Handling

For example, here are the existing steps in the exception handler for R4k platforms before adding IOS Warm Reboot:

- Step 1** Offer the exception to a registered signal handler (if available).
- Step 2** If GDB is running, pass control over to GDB. If GDB handles the exception, it will not return.
- Step 3** Write out the crashinfo file.
- Step 4** Write out core.
- Step 5** Let registered clients know that IOS is crashing.

While cold rebooting, the ROMMON saves the stack trace and the reload reason in its NVRAM region. This information is available to the user from ROMMON or from the IOS prompt (because `warm-reboot` will bypass ROMMON, we need to save the stack trace and related information in ROMMON NVRAM).

- Step 6** Return to ROMMON.

2.3.5.3.2 Changes in Exception Handling for Warm Reboot

In the sample sequence above, before any call to the service `system_exception` is made, a flag must be set via an invocation of the `wrb_set_will_reload_warm` service, to signify that a warm reboot will be attempted. This flag is checked within the functions added to `system_exception` via the `wrb_will_reload_warm()` warm reboot function, to skip any steps that make sense only when actually falling back to ROMMON. These skipped steps are put in another (possibly platform-specific) function that must be added to the warm reboot `wrb_handle_return_to_rommon` service, which is invoked when warm reboot has failed, just before returning to ROMMON.

Additionally, Step 6 will be modified to call the `wrb_handle_exception()` warm-reboot function. This function will see if `warm-reboot` needs to be initiated and if yes, initiates it. If things go wrong, control is returned back to ROMMON. For all conditional checks below, if a return value is FALSE, then control is returned to ROMMON.

2.3.5.3.3 Sample Exception Handling for Warm Reboot

If an exception occurs, the `warm-reboot` code completes the following steps:

Step 1 Checks if the `warm-reboot` storage is available (meaning that the saved copy of initialized data segment is present); only then proceeds (otherwise, returns to ROMMON) and checks if the system is crashing too soon to attempt a `warm-reboot` using the `wrb_possible()` service. Here it is determined if the system has been configured and if at least 5 minutes (or the configured time interval) have elapsed since the system was configured. The `wrb_possible()` service does the following for a warm reboot that was initiated by an exception:

- (a) Checks if the saved data segment is available for a `warm-reboot`. If not, it returns FALSE.
- (b) Checks if the system has already warm-rebooted too many times (five times is the default maximum). If so, it returns FALSE.
- (c) If none of the above, it returns TRUE.

If it returns FALSE, it prints an error and returns.

Step 2 Calls `wrb_platform_attempt_warm_reboot()` with the signal number and context to determine if the platform wants to warm reboot. The platforms can use the `wrb_is_wrb_signal()` function to determine if the signal is one of the signals that initiates a warm reboot (SIGILL, SIGBUS, SIGSEGV, SIGWDOG, and SIGRELOAD). The simplest implementation of `wrb_platform_attempt_warm_reboot()` should call `wrb_is_wrb_signal()` and return its return value.

Step 3 If the above steps from the `wrb_handle_exception` function are okay (see section “Sample Standard Exception Handling”), the `wrb_handle_exception` function calls the `wrb_reload_warm()` function to do the following:

- (a) From the data discovered in the previous steps, find out how many times the system has warm-rebooted already. Do not continue if the system has already warm-rebooted the maximum number of times allowed (this is configurable with the **warm-reboot count** and **warm-reboot uptime** commands).
- (b) Check the integrity of the `warm-reboot` header saved in the `warm-reboot` storage. Also check the saved data, text, and data checksums.
- (c) Call the `wrb_platform_reset()` platform-specific code to carry out any necessary initializations before a long jump is done. This includes disabling interrupts, disabling watchdog, and disabling port adapters/interfaces (at startup, IOS code expects that port adapters do not generate interrupts before stacks are initialized). Check if the platform-specific code executed successfully.
- (d) Jump to the start of the text segment using `long jmp`. The flow should resume in the `else` clause of the `if` statement calling `set jmp()`. Here, uncompress and then copy (after (e) below performs the `wrb_platform_pre_data_restore_init()` function) the saved data segment using `wrb_restore_data_segment()`.

- (e) Call the `wrb_platform_pre_data_restore_init()` platform-specific function before data is restored, to carry out any prerestore initialization, such as changing over from the IOS exception handlers to the system exception handlers, etc.
- (f) Call the `wrb_platform_post_data_restore_init()` platform-specific function after the data is restored, to carry out any further initialization, such as flushing caches, etc.

2.3.5.3.4 Summary of Warm Reboot on an Exception

Here is a summary of the tasks:

- 1 On an exception, check whether `warm-reboot` should be carried out. The following checks are done:
 - Check if `warm-reboot` has been enabled.
 - Check if the data segment has been saved at startup.
 - Check with the platform whether it is OK to `warm-reboot`.If any of the above checks fail, then cold-reboot is initiated.
- 2 Before initiating `warm-reboot`, check the integrity of the `warm-reboot` storage, text, and read-only data section.
- 3 Call the platform-specific functions to carry out pre-warm-reboot tasks, such as disabling the interrupts, the watchdog, and the port adapters.
- 4 Uncompress and copy the data segment from the `warm-reboot` storage to the appropriate location.

2.3.5.4 User-Initiated Warm Reboot

The user can initiate a warm reboot at any point using a variant of the **reload** command. All variants of the **reload** command will now accept an additional keyword, **warm**, after **reload**. When any of the **reload warm** commands are executed, the `warm-reboot` code completes the following steps:

- Step 1** The `reload_command()` function checks if the requested reload is a warm reload. If so, the `reload_command()` function invokes the `warm-reboot` service, `wrb_possible`, to check if it is possible to do a warm reboot.
- Step 2** The `wrb_possible()` service does the following for user-initiated warm reboot:
- (a) Checks if the saved data segment is available for a `warm-reboot`. If not, it returns FALSE.
 - (b) If it is available, it returns TRUE.
- If it returns FALSE, it prints an error and returns.
- Step 3** Let the other sanity checks of the `reload` command complete. If everything goes fine and a `reload` is confirmed, record that the requested `reload` is warm or cold using the `wrb_set_warm_reload()` service.

Step 4 Next, depending upon whether the requested reload is immediate or delayed, the `reload_system()` function is called. In this function, just before the invocation of the `hardware_shutdown` service, a flag is set via an invocation of the `wrb_set_will_reload_warm` service to signify that a warm reboot will be attempted. This flag can be checked within the functions added to `hardware_shutdown`, via the `wrb_will_reload_warm` warm reboot function, to skip any steps that make sense only when actually falling back to ROMMON. These skipped steps can be put in another function that should be added to the `wrb_handle_return_to_rommon` warm reboot service, which is invoked when warm reboot has failed, just before returning to ROMMON.

Step 5 Before returning to ROMMON via `mon_reload()`, check the previously saved value (saved in Step 3) using the `wrb_is_warm_reload` service and attempt a warm reload using the `wrb_reload_warm` service if it returns TRUE, else fall back to call `mon_reload()`.

Step 6 The `wrb_reload_warm()` function, added to the `wrb_reload_warm` service, is called to do the following:

- (a) Ignore the restrictions (configurable with the **warm-reboot count** and **warm-reboot uptime** commands) that are used upon an exception to determine whether to initiate warm reboot or cold reboot.
- (b) Check the integrity of the `warm-reboot` header saved in the `warm-reboot` storage. Also check the saved data segment, text, and read-only data checksums.
- (c) Call the `wrb_platform_reset()` platform-specific code to carry out any necessary initializations before a long jump is done. This will include disabling interrupts, disabling watchdog, and disabling port adapters/interfaces (at startup, IOS code expects that port adapters do not generate interrupts before stacks are initialized). Also, check if the platform-specific code executed successfully.
- (d) Jump to the start of the text segment using `longjmp`. The flow should resume in the `else` clause of the `if` statement calling `setjmp()`. Here, uncompress and then copy (after (e) below performs the `wrb_platform_pre_data_restore_init()` function) the saved data segment using `wrb_restore_data_segment()`.
- (e) Call the `wrb_platform_pre_data_restore_init()` platform-specific function before data is restored, to carry out any prerestore initialization, such as changing over from the IOS exception handlers to the system exception handlers, etc.
- (f) Call the `wrb_platform_post_data_restore_init()` platform-specific function after the data is restored, to carry out any further initialization, such as flushing caches, etc.

Note The `wrb_reload_warm()` function is also called from `wrb_handle_exception()`.

2.3.5.4.1 Summary of Warm Reboot through the CLI

Here is a summary of the tasks:

- 1 Execute CLI logic to enable/disable/show the `warm-reboot` state.
- 2 Execute CLI logic to reload upon the user's command.

APIs are exported to perform these tasks. The functions are part of a new subsystem, `sub_warm_reboot`, which must be included for platforms supporting `warm-reboot`.

2.3.6 Future IOS Warm Reboot Implementation

The following future implementation has been defined for IOS Warm Reboot:

- 1 If the user issues the **boot system** ... CLI to change the head of the boot system queue (to boot the system with some image other than what is currently running), on the next reload warm reboot will not be attempted, but the warm-reboot configuration will remain. A message is given to the user saying that a warm reboot is configured but the next reboot will be a cold reboot. If the **boot system** command is not the first boot system command in effect (not at the head of the queue), then warm-reboot is used, in which case the above mentioned error message is not generated.

Note As of June 2003, this had not yet been implemented.

- 2 The reload/crash context is still available after warm-rebooting (accessed through the **show stacks** command in IOS). Currently, the reload context is saved by ROMMON.

Note As of June 2003, this had not yet been implemented.

- 3 Location of the Copy of Initialized Variables: There is a trade-off for IOS Warm Reboot; some additional memory is consumed because of the need to store a copy of the initialized memory. This cannot be stored in flash because flash storage defeats the purpose of warm reboot, which is for fast reboot, so it needs to be stored in memory.

However, to reduce the memory footprint, and as an additional safety precaution, the copy of the initialized memory should be stored in a compressed form in a section of memory that has been marked RO (Read-Only) by the CPU (as of June 2003, this had not yet been implemented). This reduces the size of memory consumed for IOS Warm Reboot by about 50% while sacrificing only a few seconds of performance. Furthermore, should the copy of the initialized variables be corrupted (even with the RO setting), the file would not be uncompressible in that case, which would allow us to safely fall back to a cold reboot.

Note As of June 2003, this had not yet been implemented.

The memory set aside for IOS Warm Reboot is not lost if the user does not make use of the warm reboot feature because the memory region can be carved for use by the region manager.

2.3.7 Data Structure Support for IOS Warm Reboot

The major data structure that supports warm reboot is `wrb_saved_hdr_t`. This data structure is saved along with the restorable data in the memory and is used at the time of warm reload. The `wrb_saved_hdr_t` data structure is in `.../os/warm_reboot_private.h`. The `wrb_saved_hdr_t` data structure becomes a part of the saved data that persists across and enables warm reboots.

2.3.8 How to Make a Platform IOS Warm-Reboot-Aware

The following subsection describes how to make a platform warm-reboot-aware.

2.3.8.1 IOS Warm Reboot Example: 7200

To add the IOS Warm Reboot capability to the 7200, follow these steps:

- Step 1** Check if the target branch has warm reboot code (it would suffice to see if the `sys/os/warm_reboot.c` file is present); if not, port the changes to the following files from the latest Diffs—xxxxx enclosure from the DDTS CSCdz16687:

```
sys/machine/image_elf.h
sys/os/compress_lzw.c
sys/os/init.c
sys/os/kinit.c
sys/os/main_4k.c
sys/os/sum.c
sys/os/sum.h
sys/src-4k/gdb_4k.c
sys/ui/exec.c
sys/ui/exec_chain.c
sys/ui/exec_reload.h
sys/os/cfg_warm_reboot.h
sys/os/exec_show_warm_reboot.h
sys/os/warm_reboot.c
sys/os/warm_reboot_chain.c
sys/os/warm_reboot_api.h
sys/os/warm_reboot_private.h
sys/os/warm_reboot_registry.c
sys/os/warm_reboot_registry.h
sys/makelibs
sys/makesubsys.platform
```

See how the `WARM_REBOOT` macro has been defined in `sys/machine/cisco_c7100.h` and make the same change(s) for your platform to be able to include the stuff protected by the `WARM_REBOOT` conditional compilation flag in your image.

And for non-MIPS platforms, change `main_xxx.c` and `gdb_xxx.c` for the target processor family.

- Step 2** Find the file containing the `platform_memory_init()` function for the target platform, and make the same changes as made to `src-4k-c7100/c7100_memory.c`. See the changes made to the `platform_memory_init()` function, made for the 3660 platform in the enclosure Diffs--geo_t_pi1_itd1 to the DDTS CSCdz16687 (or see the changes made to the `platform_memory_init()` function in the latest Diffs attachment for the platform most recently made warm-reboot-aware).

- Step 3** Only the read-write initialized data segment needs to be saved and restored. The BSS is zeroed by the IOS `main()` routine. Currently, macros are available to differentiate the text, data, and BSS segment. Read-only data is packaged in as a part of the data segment. To avoid saving the read-only data in the warm-reboot storage, distinguish the read-only data from the read-write data. This can be done by changing the linker script to place the read-only data in the text section. In the object directory, make changes to the linker script, as shown below.

For example, for the 7200, the script will be `obj-4k-c7100/predator.link`, as shown here:

```
SECTIONS {
    .text . : {
        *(.text)
        *(gnu.linkonce.t*)
        . = ALIGN(0x20000);
```

```

        _etext = .;
        _etext = .;
    }
.rodata . : {
    _fdata = .;
    *(.rodata)
    *(.gnu.linkonce.r*)
    . = ALIGN(32);
}
.data . : {
    _rwdatal = .;           /* this is added */
    *(.data)
    *(.gnu.linkonce.d*)
    . = ALIGN(32);
}
_gp = (. + 0x8000);
.sdata . : {
    *(.sdata)
    *(.gnu.linkonce.s*)
    . = ALIGN(32);
    edata = .;
    _edata = .;
}
_sbss . : {
    __bss_start = .;
    _fbss = .;
    *(.sbss) *(.scommon)
    . = ALIGN(32);
}
.bss . : {
    *(.bss)
    *(COMMON)
    . = ALIGN(32);
    _end = .;
    end = .;
    _wrb_storage_start=.; /* <= start of warm-reboot storage */
    _wrb_storage_end=.;   /* <= end of warm-reboot storage */
}
}
}

```

Note Because in this example the warm-reboot storage lies after the end of BSS, a hole need not be reserved. That's why both the start and end are adjacently placed. The warm-reboot code carves out the required space from the beginning of the heap. The platform memory code has to adjust the start of the heap accordingly.

- Step 4** In the `src-xx-yyyy` directory for the target platform, implement the functions required by warm reboot in the `xxxxxx_wrb_support.c` file, and if required, a `xxxxxx_wrb_support.h` header file can also be created. For the 7200, this is in `src-4k-src7100/c7200_wrb_support.c` and `src-4k-c7100/c7200_wrb_support.h`.

The functions that IOS Warm Reboot support expects the platform to provide are:

- `boolean wrb_platform_pre_data_restore_init(void)`

This function carries out any platform-specific tasks before the data segment is copied over.

- `boolean wrb_platform_post_data_restore_init (void)`

This function carries out any platform-specific tasks after the data segment is copied over.

- `boolean wrb_platform_attempt_warm_reboot (int signal_no, int signal_code, void *context)`

This function returns TRUE if IOS should warm-reboot. Your platform must call `wrb_is_wrb_signal()` to pass the signal number and return its return value. Your platform can make more intelligent decisions on whether to warm-reboot or to cold-reboot inside this function.

- `void wrb_platform_return_warm_reboot_storage (void)`

This function returns the warm reboot storage to the heap when warm reboot is disabled.

- `boolean wrb_platform_disallow_config (void)`

At the time of updating this document, the 7200 platform does not implement this function.

This constitutes the major effort in making a platform warm-reboot-aware, because the implementation of these functions requires the platform to explore exactly what needs to go into each of these functions.

In addition to this, the platform needs to revisit the functions added to the `system_exception` and `hardware_shutdown` services to see if any of the logic there assumes that ROMMON will take control and do something for the platform, because this assumption cannot be made for warm reboot. A way of doing the same in IOS may need to be created on a per-platform basis (for an example, see the 3660 or 3745 platform). If these steps only make sense when returning to ROMMON, they should be skipped based on the return value of the `wrb_will_reload_warm()` function. These skipped steps will need to be completed in a separate (possibly platform-specific) function, and this separate function will need to be added to the warm reboot `wrb_handle_return_to_rommon` service.

Step 5 In the object directory, change the make file to include the following files in the make file's `PLATFORM_OTHER` option:

- The subsystem `sub_warm_reboot.o` file.
- The `xxxxx_wrb_support.o` file from the file `xxxxx_wrb_support.c` created in Step 4.

2.4 IOS Warm Upgrade

This section includes a description of the IOS Warm Upgrade facility and instructions about using the IOS Warm Upgrade API. For more details on the design of IOS Warm Upgrade, see the “*Warm Upgrade Software Functional Specification*”, [EDCS-333055](#).

For more information on 3750E and 3560E warm upgrade functions, see the “*PIXAR Warm Upgrade Software Functional and Design Specification*”, [EDCS-602256](#).

IOS Warm Upgrade was developed to provide a facility for upgrading/downgrading IOS with smaller downtime. IOS Warm Upgrade is available in IOS Versions 12.3T and 12.2S. The reduction in downtime results from the fact that while the new image is being downloaded and decompressed, the current image continues to function normally.

Warm Upgrade is a facility by which an IOS image can read in and decompress another IOS image and transfer control to it. This reduces the downtime of a platform for a planned upgrade/downgrade. This facility is complementary to Warm Reboot and is available in conjunction with Warm Reboot. Warm Reboot provides a facility to quickly reboot a platform with the same image that is currently running. How is Warm Upgrade complementary to Warm Reboot? Warm Reboot can achieve smaller downtime but has the limitation that the currently running image can only reload itself. It's not possible to load a different image. Warm Upgrade takes care of this and provides a way to change the IOS version without incurring as much downtime as is required by a cold reboot.

Currently, to upgrade/downgrade IOS, control is transferred to ROMMON. ROMMON, along with the help of the bootloader, carries out the required action. While this is in progress, the device is down. Warm Upgrade brings in gains because, while the new IOS image is read and decompressed, packet forwarding can continue. Only when we overwrite the current IOS with the next, do we need to stop all activities until the new image loads and configures the system afresh.

Because Warm Upgrade uses the platform API functions developed for IOS Warm Reboot, it is packaged into images that have Warm Reboot support. Currently, only 7200 images have this support on 12.2S, and 3745, 3660 and 2800 have it on 12.3T. The warm upgrade available on 3750E and 3560E is disabled by default and is not currently supported for the customers. Use the **service internal** command to enable it.

IOS Warm Reboot is possible only if there is enough free memory in the system to hold a decompressed IOS image. This facility is available only in conjunction with IOS Warm Reboot. This facility is only provided for ELF images to begin with. The support can be extended later for other image formats.

This IOS Warm Upgrade documentation includes the following subsections:

- IOS Warm Upgrade Components
- IOS Warm Upgrade API Functions
- IOS Warm Upgrade CLI
- IOS Warm Upgrade Flow Overview

2.4.1 IOS Warm Upgrade Components

The following list includes the components of the IOS Warm Upgrade feature:

- IOS Warm Upgrade Subsystem

Warm upgrade subsystem carries out the tasks which, for a normal reload are done in part by:

- IOS reload command handler
- ROMMON
- bootloader

- Czip Decompression Routines

These are already available in the czip image. They have been pulled into IOS and made a part of ukernel.

- IOS routine to relocate image and jump to entry point (`copy_and_launch` for MIPS)

This is currently not included in an IOS image. However, for a Warm Upgrade-aware image, this needs to be packaged into the image. `copy_and_launch` needs to be packaged into an IOS image by adding the `asm_wrb.o` file in the platform makefile along with the other Warm Reboot files.

- Image Verification APIs

These are used for Warm Upgarde to verify an image before upgrading.
- NVRAM and ROMMON

NVRAM (and ROMMON) pass information from one IOS instance to the next. For Warm Upgrade, they pass the same information from the currently running IOS to the next which they pass in the case of a normal reload.
- Warm Upgrade CLI

The Warm Upgrade CLI is an offshoot of the **reload** privileged exec command. This CLI supports all the options of the **reload** command except the **file** option on the 3750E and 3560E images.

2.4.2 IOS Warm Upgrade API Functions

Warm Upgrade does not export an API. Neither does it expect the platform to export any APIs apart from those discussed in section “IOS Warm Reboot.” To extend the Warm Upgrade facility to your platfrom, just follow the steps outlined in section “How to Make a Platform IOS Warm-Reboot-Aware” to make it Warm Reboot-aware. Warm Upgrade will automatically become available for your platform. To extend Warm Reboot/Upgrade to new processor types (currently supported on MIPS), please contact the OS/Infra team.

2.4.3 IOS Warm Upgrade CLI

There are two parts to the IOS Warm Upgrade CLI: modifications to the existing IOS Warm Reboot CLI and a new CLI.

2.4.3.1 Modifications to Existing CLIs

You can affect a warm upgrade/downgrade using the following privileged exec CLI:

```
reload [/verify | /noverify] warm file url [debug]
```

The last **debug** keyword is a service internal extension to the command to allow the new IOS image to break into the debugger upon reload.

Additionally, the **show reload** CLI can be used to see the particulars of a delayed reload.

The **file** option is disabled for the 3750E and 3560E images because of the stacking nature of the 3750E. To perform an warm upgrade on these platforms each box’s BOOT environment variable must specify that box’s targeted image, that is, the image for which the warm upgrade is to be loaded.

2.4.3.2 New CLI

A new privileged exec CLI has been implemented to print warm reboot-related debug information:

```
debug warm-reboot
```

2.4.4 IOS Warm Upgrade Flow Overview

The Warm Upgrade functionality is divided into the following code pieces. Given below is a list of tasks performed by each. This provides an idea of the Warm Upgrade system flow too.

2.4.4.1 Common Warm Upgrade Code

- Step 1** Take in the Warm Upgrade CLI from the options and see whether a reload is requested now or later and remember it.
- Step 2** From the options and the global configuration, find out whether a verification is required. If so, do it. (Currently done by the **reload** command.)
- Step 3** Invoke the `component_reload_prepare` service to let the components know of the impending reload. If the reload is delayed, schedule it, else go ahead. (Currently done by the **reload** command.)
- Step 4** At the reload epoch, start the reload process. Read in the image and pass on control to the image format specific warm upgrade code to understand the image format and decompress it. Except for the 3750E and 3560E, the decompression is handled by common code although it is invoked from image-specific code.
- Step 5** Up to this point, if there is any failure, we clean up and let the current IOS image continue. If there is any failure after this point, we fall back to ROMMON.
- Step 6** Populate the `boot_info` structure. (Currently done by the bootloader.)
- Step 7** We reach here only if everything is fine and we are ready for the reload. Call the `component_reload_now` service, and proceed to save the reload reason.
- Step 8** Set a flag to indicate that a Warm Upgrade will follow and ROMMON will not take control. Having done this, invoke the `hardware_shutdown` service.
- Step 9** Transfer control to platform-specific code to reset the platform. If this happens successfully, attempt the `copy_and_launch` routine.

2.4.4.2 Processor-Specific Warm Upgrade Code

- Step 1** The `copy_and_launch` routine and related functionality, like setting up parameters for `copy_and_launch`, are processor-specific.

2.4.4.3 Platform-Specific Warm Upgrade Code

Both of the steps below are done for Warm Reboot and are available for Warm Upgrade for free but are mentioned here for the sake of completeness.

- Step 1** When the `hardware_shutdown` service is invoked, check if a warm upgrade is about to happen. If so, do not rely on ROMMON to do any clean-ups and initializations that happen in the case of a normal reload and take platform-specific compensatory actions.
- Step 2** When requested by the Warm Upgrade common code, before the `copy_and_launch` routine, reset the platform. Basically the same routines which reset a platform for a Warm Reboot will be used here.

2.4.4.4 Image Format-Specific Warm Upgrade Code (only ELF is supported currently)

The Warm Upgrade common code calls image format-specific code after the image has been read in. It carries out the following tasks:

- Step 1** Check the magic and checksum of the image. Carry out the image integrity check.
- Step 2** Peek into the image format-specific headers to find out the compressed and decompressed sizes etc.
- Step 3** Call common code to decompress the image.

- Step 4** Call the image format-independent API to arrange the image contained in scattered memory blocks into ascending order of physical address (called postprocessing). This is required so that no part of the image still to be copied is overwritten by the `copy_and_launch` routine.
- Step 5** Look at the section information and build the section array to be used by the `copy_and_launch` routine. Request the processor specific code to populate the `imageinfo` structure for use by the `copy_and_launch` routine.
- Step 6** In the block with the highest address, as determined by Step (4), copy the `copy_and_launch` routine and the parameters (`section_array, imageinfo`) to be passed to it.
- Step 7** Pass control to the `copy_and_launch` routine.

2.5 Retrieving System Information

You may sometimes want to retrieve system information programmatically, that is, not through a CLI command but rather through looking at the Cisco IOS code. The following is an example of that:

Example: is there any way to obtain the information contained in **show version** programmatically in IOS?

Specifically in the following fields:

- platform
- current boot location
- IOS version

Yes, there is.

- The platform name can be obtained by looking at
`platform_get_string(PLATFORM_STRING_NOM_DU_JOUR)`
- The boot location can be obtained from the global variable `sysconfig.url`
- The version can be contained in the global character string `version`.

P A R T 2

Kernel Services

Basic IOS Kernel Services

Redrew Figure 3-1, Figure 3-2, Figure 3-3, Figure 3-4, Figure 3-5 and Figure 3-6. (August 2011)

Added more information about process_may_suspend() in the Table 3-2 “Functions for Checking and Suspending Tasks”. (August 2010)

Added more information about preemptive process in the section 3.3.6.3.1 Pseudo-preemption Infrastructure Memory and Performance Considerations. (July 2010)

Added the following macro to create_watched_priority_semaphore() and create_watched_recursive_semaphore() in the section 3.4.3.6.1 (July 2010)

Updated process_may_suspend_on_quantum(), process_suspend_if_req(), and process_time_exceeded() in Table 3-2 “Functions for Checking and Suspending Tasks” to say that the use of these functions is strongly discouraged. (July 2009)

3.1 Introduction

The scheduler, Process Management, and Event Management comprise the Cisco IOS Scheduler. The reason for the new sub-division is that the Cisco IOS “Scheduler” has grown large enough to be broken into three smaller more manageable modules. This chapter discusses those modules, plus additional services in the following sections:

- Scheduler
- Process Management
- Event Management
- Random Number Generation

Note Cisco IOS kernel services and scheduler development questions can be directed to the interest-os@cisco.com and pkt_scheduling@cisco.com aliases.

Note In previous editions of this chapter, the terms “process” and “task” were used interchangeably for an entity that can be scheduled in Cisco IOS software. The term “task” was also used to refer to the work done by the entity. There was some ambiguity built into this nomenclature, not just because two words were being used for the same entity, but because the dual meaning of “task” evoked an additional entity, one created by a process. That notion was reinforced by the industry’s definition of “process” as possessing one or more “threads”. From there, it was easy to suppose that each process had its own protected memory as in virtual memory systems.

In Cisco IOS, each independently schedulable entity does a single piece of work and shares nonvirtual memory with all the other schedulable entities. If an IOS feature needs to accomplish multiple tasks, the developer creates an independently schedulable entity for each task and the only relationship between the entities is in the mind of the developer. Any of the entities can interact and operate on another entity's data.

Therefore, in the current edition of this chapter (June 2002), "task" has been selected as the term for a schedulable entity in IOS. You will still come across "process" in legacy expressions, such as "process ID" (PID), and it is in the code. When you come across "process" in other IOS documentation, please understand it as a synonym for "task".

3.1.1 Terminology

To help in understanding the discussion, the following list of terms used in this chapter is provided.

atomic lock

A flag that can be set or cleared via an uninterruptable action. See also managed semaphore, semaphore, and watched semaphore.

bit field

A contiguous array of binary digits (bits) that have individual significance.

blocking

A task blocks when it waits for an event. Except for the implicit blocking that occurs when interrupt service is performed, all blocking in IOS is explicitly requested by the caller.

Boolean

Memory location that holds one of two values, TRUE or FALSE (that is, the value 1 or 0, respectively). See also managed Boolean.

dead queue

Scheduler queue for processes that have exited, but on which the schedule has not yet performed a postmortem analysis.

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.

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.

managed

An adjective used by IOS Event Management Services to identify a category of object that participates in Event Management. See also managed Boolean, managed queue, managed semaphore, and managed timer. Synonymous with watched.

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 an atomic lock 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, atomic lock.

message

In the scheduler, a simple interprocess communication (IPC) mechanism that allows two processes to communicate.

PID

Process identifier.

priority

Order in which the scheduler executes processes. Processes can run at one of the following priority levels: critical, high, medium, or low.

process

A collection of programs and/or data often arrayed under one or more "threads" or "tasks" that can be independently scheduled by the operating system. Historically, "process" was a synonym for task or thread, but this usage has been incorrect since the advent of virtual memory. In systems with virtual memory, a process occupies its own private address space. There is no virtual memory in Cisco IOS (each scheduled task shares memory with all the other scheduled tasks in IOS).

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.

ready queue

Scheduler queue used for processes that are ready and waiting to run.

semaphore

Memory location that is used by multiple processes to serialize their access to one or more resources. The resource can be anything, for example Flash memory or the table of IP routes. See also managed semaphore, atomic lock, watched semaphore.

SMP

Symmetric Multi-Processor. Where more than one CPU core shares a single system image and where any CPU can perform any operation in the system.

task

A unit of work recognized by the IOS scheduler as potentially needing system resources.

text segment

The text segment contains all the executable code (assembler instructions) contained in an image.

watched

See managed.

watched Boolean

See managed Boolean.

watched queue

See managed queue.

watched semaphore

See managed semaphore. See also semaphore, atomic lock.

3.1.2 List of Functions

Basic process functions to relinquish the processor include:

- 1 `process_suspend()`
- 2 `process_suspend_on_quantum()`
- 3 `process_may_suspend()`
- 4 `process_may_suspend_on_quantum()`

Event Management provides the following functions to manage queues:

- 1 `create_watched_queue()`
- 2 `process_set_queue_minor()`
- 3 `process_watch_queue()`
- 4 `process_set_queue_first_only()`
- 5 `process_enqueue()`
- 6 `process_unqueue()`
- 7 `process_requeue()`
- 8 `process_insqueue()`
- 9 `process_enqueue_pak()`
- 10 `process_requeue_pak()`
- 11 `process_data_enqueue()`
- 12 `process_dequeue()`
- 13 `process_data_dequeue()`
- 14 `process_peek_queue()`
- 15 `process_queue_size()`
- 16 `process_queue_max_size()`
- 17 `process_resize_queue()`
- 18 `process_is_queue_full()`
- 19 `process_is_queue_empty()`
- 20 `delete_watched_queue()`

Managed Booleans are supported with the following functions:

- 1 `create_watched_boolean()`
- 2 `process_set_boolean_minor()`
- 3 `process_watch_boolean()`
- 4 `process_set_boolean_first_only()`
- 5 `process_set_boolean()`
- 6 `process_get_boolean()`
- 7 `delete_watched_boolean()`

Managed bit fields are supported by the following functions:

- 1 `create_watched_bitfield()`

```
2 process_set_bitfield_minor()
3 process_watch_bitfield()
4 process_set_bitfield_first_only()
5 process_get_wakeup()
6 process_get_bitfield()
7 process_set_bitfield()
8 process_clear_bitfield()
9 process_keep_bitfield()
10 delete_watched_bitfield()
```

Managed semaphores are supported by the following functions:

```
1 watched_semaphore_get_name()
2 watched_semaphore_get_owner()
3 create_watched_semaphore()
4 create_watched_recursive_semaphore()
5 create_watched_priority_semaphore()
6 process_set_semaphore_minor()
7 process_lock_semaphore()
8 process_unlock_semaphore()
9 process_set_semaphore_first_only()
10 delete_watched_semaphore()
```

Atomic locks are supported by the following functions:

```
1 lock_semaphore()
2 unlock_semaphore()
```

Read/write locks are supported by the following functions:

```
1 create_watched_rwlock()
2 delete_watched_rwlock()
3 process_rwlock_rdlock()
4 process_rwlock_rdlock_timed()
5 process_rwlock_unlock()
6 process_rwlock_wrlock()
7 process_rwlock_wrlock_timed()
```

Managed messages are supported by the following functions:

```
1 process_send_message()
2 process_get_wakeup()
3 process_get_message()
4 process_watch_watched_message()
5 process_get_watched_message()
```

Managed timers are supported by the following functions:

```
1 process_sleep_for()
2 process_sleep_on_timer()
```

- 3 process_sleep_periodic()
- 4 process_sleep_until()
- 5 process_safe_may_suspend()
- 6 process_safe_sleep_on_timer()
- 7 process_safe_sleep_periodic()
- 8 process_safe_sleep_until()
- 9 process_watch_mgd_timer()
- 10 process_watch_timer()

To wait until events occur, use the following functions:

- 1 process_wait_for_event()
- 2 process_wait_for_event_timed()

Here is a list of the "safe" blocking calls:

- 1 process_safe_may_suspend()
- 2 process_safe_sleep_on_timer()
- 3 process_safe_sleep_periodic()
- 4 process_safe_sleep_until()
- 5 process_safe_suspend()
- 6 process_safe_wait_for_event()

3.2 Scheduler

The task scheduler is responsible for scheduling and executing kernel processes on a CPU. Because the scheduler is run-to-completion, all tasks must voluntarily relinquish control to the scheduler. In general, a task should perform a small amount of work, relinquish the processor, and then continue working the next time it receives the CPU from the scheduler. Basic functions to relinquish the processor unconditionally include:

- process_suspend()
- process_suspend_on_quantum() (calls process_suspend() if process has used up its processing quantum)

Functions to relinquish the processor conditionally (for example, only if there are any higher priority processes waiting to be scheduled) include:

- process_may_suspend()
- process_may_suspend_on_quantum() (calls process_may_suspend() if process has used up its processing quantum)

It is important to note that, because the scheduler is run-to-completion, it cannot guarantee that all tasks will receive sufficient CPU resources to guarantee that they respond in real time. In fact, such a guarantee is the collective responsibility of all of the tasks in the system. Real-time behavior is achieved by not overloading the system, frequent yielding of the CPU by all tasks, and the judicious use of task priority.

This means that you need to design software that operates in a time sliced manner. The time slices need to be small enough to allow the scheduler to respond to the needs of all tasks. If a more responsive system is desired, the scheduler must run more often to allow the chance/opportunity to schedule higher priority tasks.

Developers interact directly with the scheduler through the Process Management and Event Management modules.

3.2.1 Cisco IOS Task States

After task initialization, a task primarily spends its life in one of four states: idle, ready-to-run, running and dead state. When a Cisco IOS task is created it begins its life in the idle state and is then moved to the ready-to-run state. Transitions between these states may be summarized as follows:

- Idle to ready-to-run: This occurs when a managed object changes state or a timer expires.
- Ready-to-run to running: This occurs when the scheduler selects the task to execute.
- Running to ready-to-run: Known as “suspension”, or “yielding the processor”, this occurs when the task has more work to do but voluntarily yields control anyway.
- Running to idle: This occurs when the task must wait for a managed object to change state or a timer to expire.
- Running to dead: This occurs whenever the task self terminates or is terminated by the scheduler. The latter is usually caused by a “watchdog timer event”.
- Idle (or ready-to-run) to dead: This occurs because another task has issued the `process_kill()` function. A task in idle state is never moved directly to the dead state. When one task kills another task, it alters the other task’s execution context to `process_last_gasp()`, and then moves the process to the ready state. The scheduler gives it a chance to run and it runs `process_last_gasp()` in its own (dying task) context. It is given a chance to execute its `SIGEXIT` handler, to cleanup the resources it may have, and then it is moved to the dead state.
- There is a fifth state called the corrupted state. This is a special state where the scheduler tests for memory corruption. If any is found in the Cisco IOS task structure, the Cisco IOS task is removed for scheduling and placed on the corrupt queue for debugging purposes.

3.2.2 Scheduler and IOS Tasks

When the scheduler is ready to begin executing the next task, the scheduler saves its main loop registers and stack information before restoring the registers and stack information for the next task to be run. Once the next task’s context has been restored, control of the processor is returned to the next task.

3.2.3 Cisco IOS Task Scheduling Characteristics

There is rarely contention between tasks because each task runs to completion and, therefore, finishes with resources before entering the suspend or idle states. However, contention can exist between a task and an interrupt service routine. In order to eliminate or minimize such contention, tasks may disable interrupts during the critical section of code and then re-enable interrupts when the critical section of code is done.¹

3.2.4 Scheduler Queues

The scheduler manages tasks using three types of queues.

- Ready queues

1. In a cooperative multitasking environment on a uniprocessor platform, disabling interrupts is often the preferred method of preventing interrupt contention, largely because it is much more efficient.

- Idle queues
- Dead queues

These are implemented using the Cisco IOS List Manager. More information on the Cisco IOS List Manager may be found in the “Queues and Lists” chapter. A given task appears on one and only one of these queues at a given instance. There is also a one to one correspondence between the state of the task and the queue on which it resides.

The scheduler maintains one list per task priority or task type. This allows it to easily determine which task should be scheduled next. The kernel identifies each task by its task number, called a task identifier (process ID, or PID).¹

3.2.4.1 Ready Queues

Ready queues represent tasks that are ready to execute, but that have not been scheduled. There is one ready queue for each task priority: critical, high, normal, and low. When a Cisco IOS task receives the event it is waiting for, the synchronization code tells the scheduler to move the task from the idle queue to the appropriate ready queue or that a task is now ready-to-run. For example, the code fragment that enqueues a packet on the VINES managed queue for receiving packets is responsible for waking up the VINES task. The VINES task does not poll the managed queue at every pass of the scheduler. The event that causes the VINES task to be ready to begin executing again can be the expiration of a timer, or an asynchronous event, such as a packet being enqueued into a specific queue.

For more details, see section 3.2.5, “Scheduler Selection Algorithm.”

3.2.4.2 Idle Queues

The idle queue represents tasks that are waiting for the completion of one or more events. Such events are represented by managed objects or timers. The scheduler only has one idle queue. Cisco IOS tasks are moved to this queue by Event Management when they need to synchronize to an event such as a change in the state of a managed object, or a timer’s expiration. The sleep operation could be thought of as waiting for a timer to expire.

All tasks 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.

3.2.4.3 Dead Queues

The dead queue represents tasks that have been terminated and are awaiting postmortem analysis and cleanup by Cisco IOS Process Management. The scheduler has only one dead queue. A lot of this is Process Management’s responsibility. The scheduler only helps out by providing Process Management the opportunity. When the scheduler processes the dead queue, it does the following:

- Adds to dead task memory accounting, this task’s allocated and freed memory accounting, as well as the associated packet buffers’ allocated and freed memory accounting.
- Frees all implicitly allocated memory used by the task. This includes the stack, process structure, and managed object structures.
- Returns the unique PID to the pool of available PIDs.

If this task is being debugged, it is left on the dead queue and the above work is not performed.

1. The acronym PID is a legacy term.

3.2.4.4 Moving Tasks between Queues

Tasks are constantly moved from one queue to another but are always linked to one of the scheduler's queues. When a task is moved between queues, the scheduler has to consider the following points:

- Interrupts are disabled.
- A running task is unlinked from its ready queue only when it calls a kernel interface. This causes the task to suspend, which signifies that it can be safely moved.
- If a running task requests to sleep or wait and, during the course of its last time slice, new events have occurred that would make the task executable, the task is not placed on the idle queue; instead, the task remains on the ready queue. This protects against the race condition that can occur if a task is relinquishing the processor at the same time that an interrupt-level code path enqueues data for the task. This condition is specific to Event Management and how the scheduler interacts with Event Management. This condition can happen when the `process_wait_for_event()` or `process_wait_for_event_timed()` functions are called. The check for the race condition in any other functions has not been found. Functions such as `process_suspend()`, `process_may_suspend()`, and related functions (such as `process_suspend_on_quantum()`) simply yield the processor, so checking for events and the race condition are unnecessary because the tasks are never moved from the ready queue.

3.2.5 Scheduler Selection Algorithm

The scheduler selection algorithm highly favors tasks dealing with packet forwarding. This is by design based on the mission of Cisco IOS software (optimize the sending, receiving and forwarding of packets between network interfaces). While some operating systems use a weighted design such that all tasks are guaranteed a portion of the processor, Cisco IOS software focuses on those tasks that perform packet forwarding to the detriment of other tasks. This is implemented primarily through the use of interrupt service routines and task priority selection.

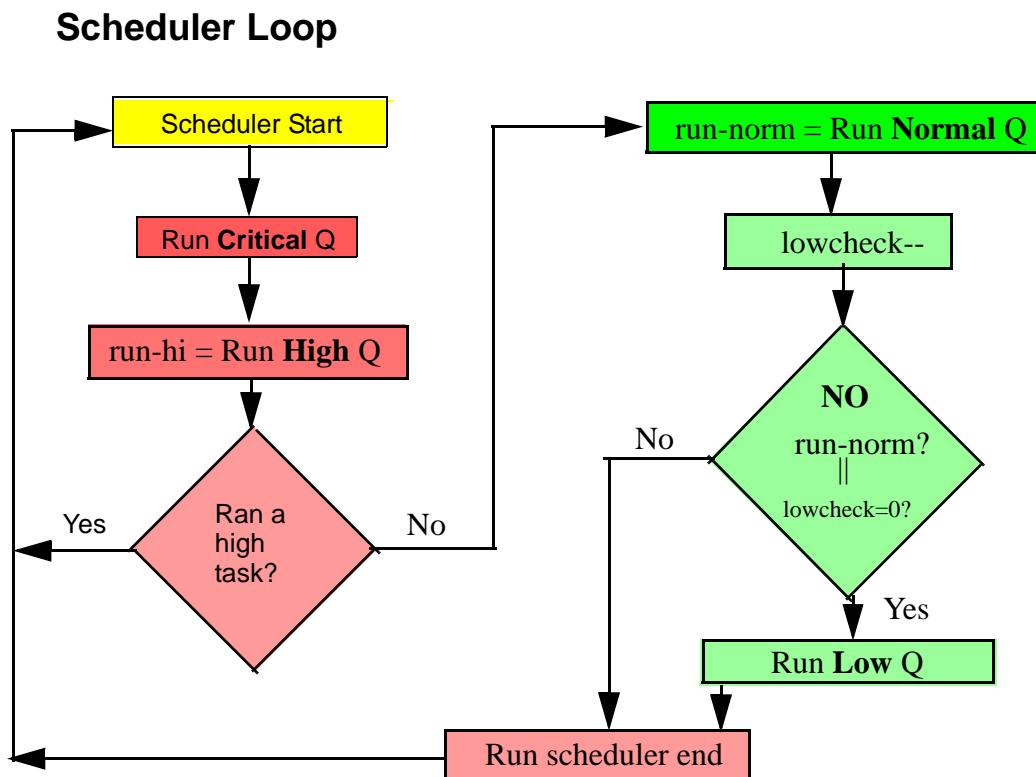
The scheduler loop is as follows.

These are the steps:

- 1 Start the scheduler loop.

Figure 3-1 illustrates the operation.

Figure 3-1 Operation of Scheduler Queues

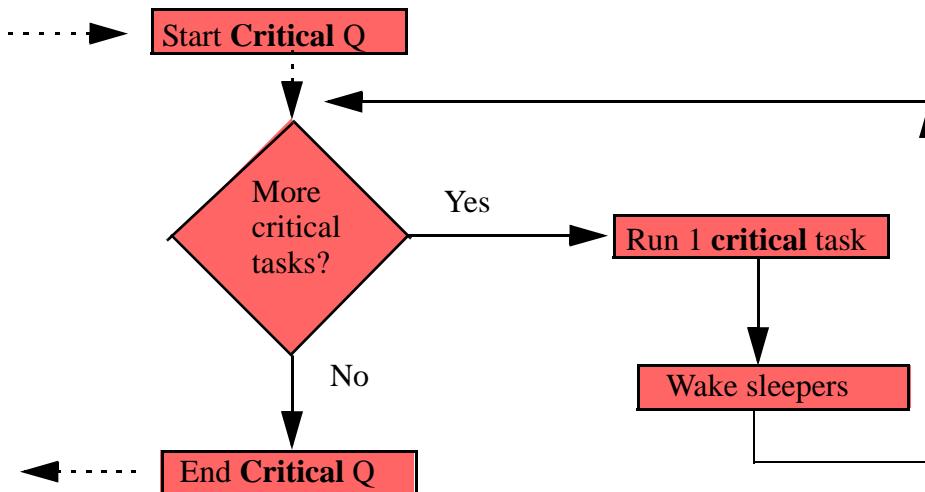


- 2 Run the critical priority queue.
 - (a) Check if there are any critical priority tasks.
 - (b) If there are, run one critical priority task, wake any sleeping tasks and check to see if there are more critical priority tasks.
 - (c) If not, run the high priority queue.

Figure 3-2 illustrates running the critical queue.

Figure 3-2 Run Critical Queue

Scheduler “Run Critical”

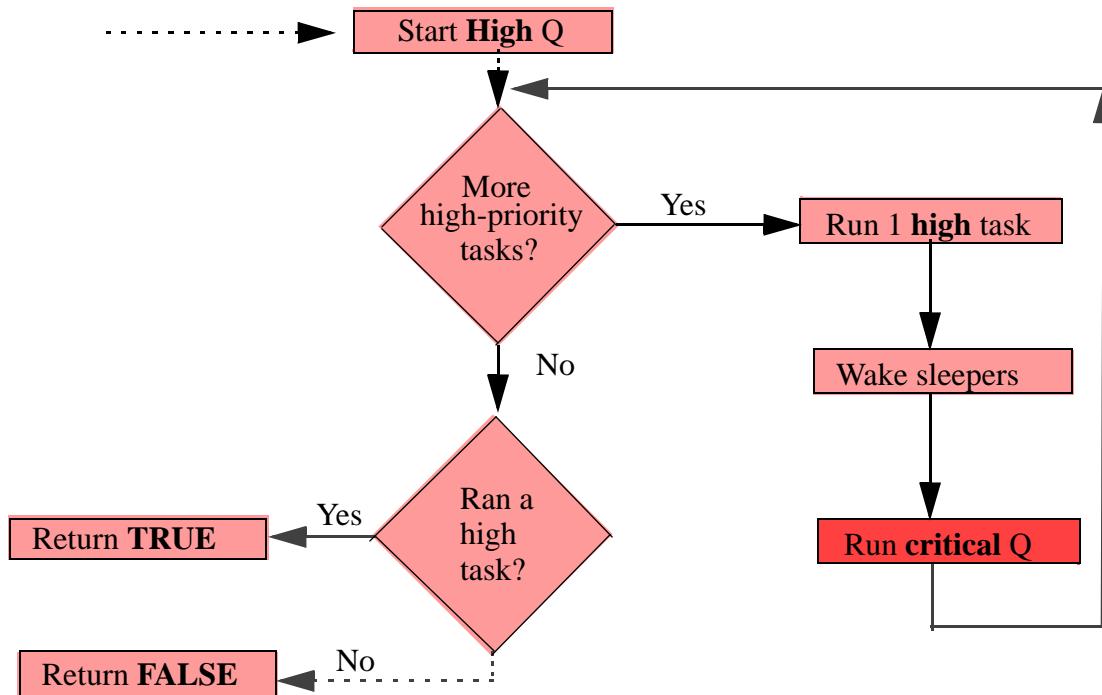


3 Run the high priority queue.

- (a) Check if there are any high priority tasks.
- (b) If there are, run one high priority task, wake sleeping tasks, run the critical queue and check to see if there are more high priority tasks.
- (c) If not, check to see if a high task has run. If it has, run the critical priority queue. If it has not, run the normal priority queue.

Figure 3-3 illustrates running the high priority queue.

Figure 3-3 Run High Queue

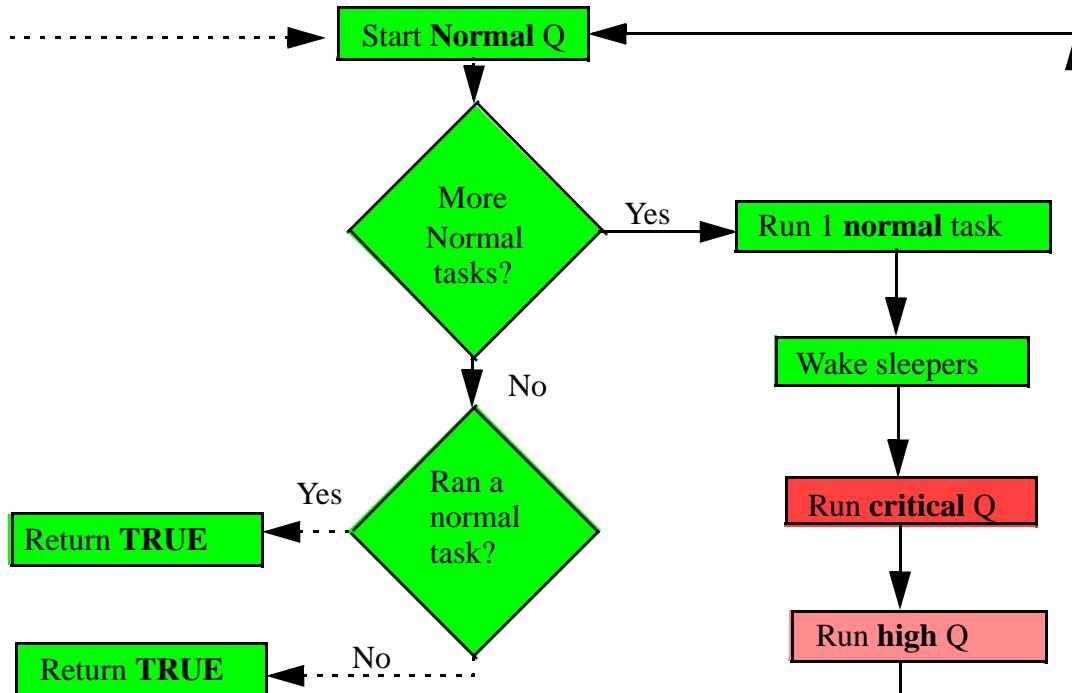
Scheduler “Run High”

4 Run the normal priority queue.

- (a) Check if there are any more normal priority tasks.
- (b) If there are, run one normal priority task, wake any sleeping tasks, run the critical priority queue, run the high priority queue and check to see if there are any more normal priority tasks.
- (c) If there are no normal priority tasks, run the low priority queue. If there are *continually* normal priority tasks, slip the low priority queue in every low count time through the loop. If there are always normal priority tasks, every low count time, ignore the normal priority tasks for awhile and run all the low tasks, intermixed with the critical/high priority tasks. Then return to processing normal priority tasks.

Figure 3-4 illustrates running the normal priority queue.

Figure 3-4 Run Normal Queue

Scheduler “Run Normal”

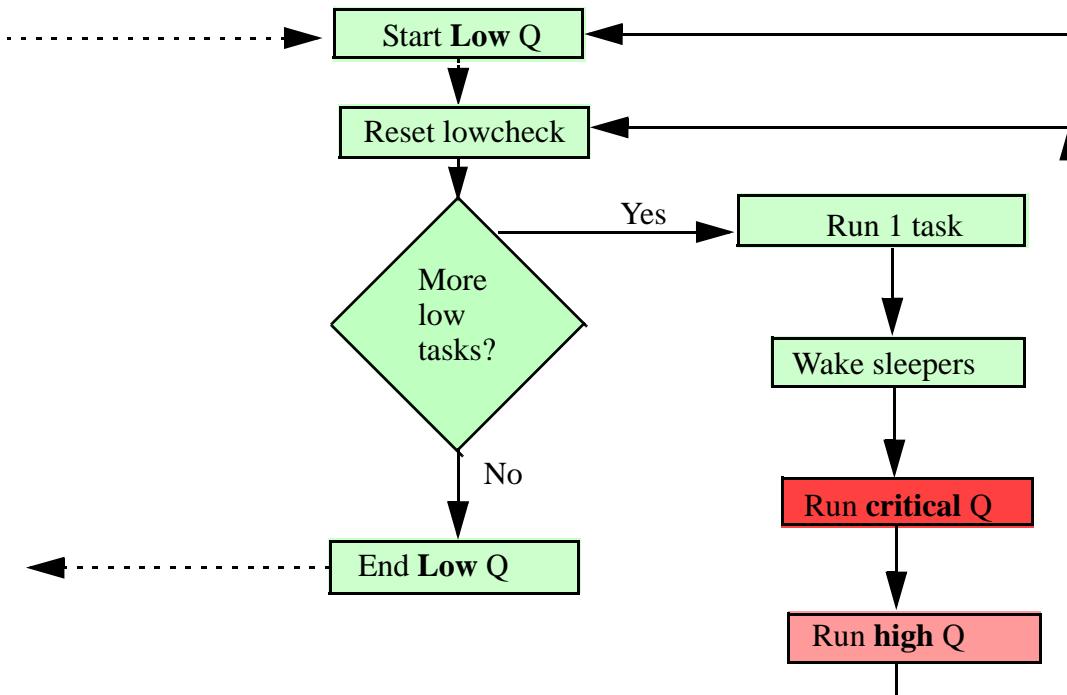
5 Run the low priority queue.

- (a) Reset the lowcheck variable.
- (b) Check if there are any more low tasks.
- (c) If yes, run 1 low priority task, wake sleepers, run the critical priority queue, run the high priority queue, and then check to see if there are any more low tasks.
- (d) If not, go to the end of the scheduler loop.

Figure 3-5 illustrates running the low priority queue.

Note An important note is the Low Q function only runs the number of tasks it detects at the beginning of the function. If additional low priority tasks get scheduled to run, they will not be executed this iteration.

Figure 3-5 Run Low Queue

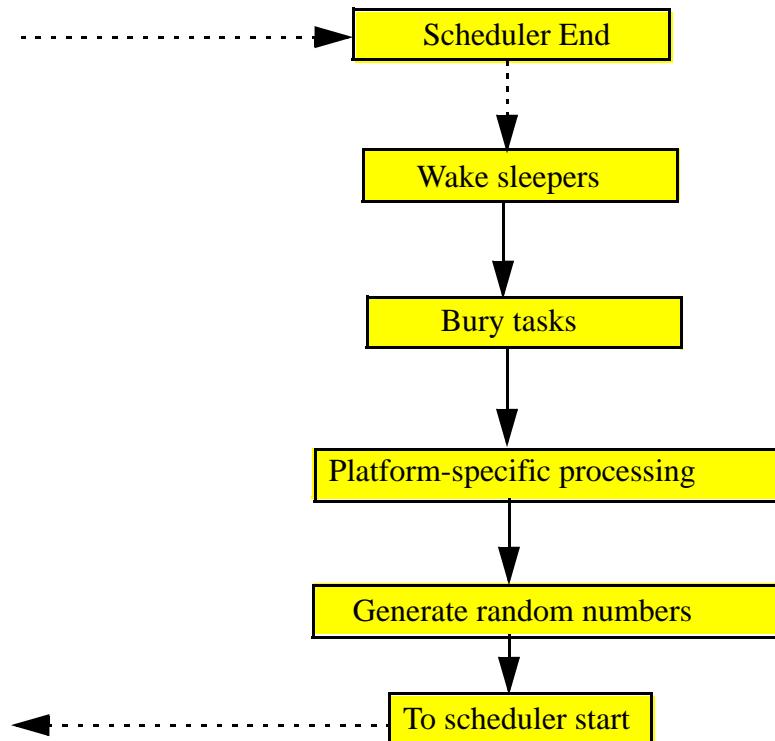
Scheduler “Run Low”

- 6 End the scheduler loop.
- Wake any sleeping tasks.
 - Bury any “dead” tasks.
 - Call the specified platform-specific routine, if established via `kernel_set_sched_loop_hook()`.
 - Generate random numbers.
 - Return to the beginning of the scheduler loop.

Figure 3-6 illustrates the ending of the scheduler loop.

Figure 3-6 Scheduler Loop End

End of Scheduler Loop



3.2.6 Important CLI Commands

Here is an example of how to use the scheduler commands. Scheduler configuration commands are available only when operating in “configuration mode”:

```
ios (config) # process-max-time ms
```

Here are the scheduler commands:

- **heapcheck process**

Validates all memory headers and linkages *after each task gives up control*. Normally this is done each minute. This could severely degrade performance and should only be performed in a production environment with consensus of Cisco TAC.

- **heapcheck poll**

Same as **heapcheck process** for tasks still using older-style scheduler. Same functionality and recommendations.

- **process-max-time**

This is a system-wide variable that indicates the recommended maximum time (in milliseconds) to use the processor before giving another task the chance to run.

This variable is used in `process_may_suspend()`, `process_get_info()`, and `process_set_info()`. Historically, the `process_max_time()` function was set to a default of 200ms for all platforms. Recently, a few platforms have overridden this with a platform-appropriate default. For example, the Catalyst 4000 uses 20ms. Check your particular codebase.

- **max-task-time 200**

The amount of time in milliseconds a task can run before signalling it is “hogging” the CPU, which defaults to 2000 ms. Also, the default behavior is to kill the process only if it is fatal (if the task gets another watchdog within 12 hours since the first watchdog, then it is fatal), and a handler has been registered. Do not modify without consensus of Cisco TAC.

- **scheduler run-degraded**

If the scheduler finds an internal error, such as a corrupted process structure, by default Cisco IOS software will force a crashdump. If **scheduler run-degraded** has been set, the scheduler will remove the corrupted process and continue running. Depending on the process removed, the network device may not function anyway (for example, if the IP process is removed).

- **scheduler process-watchdog**

This finds the default action of a watchdog timeout.

3.3 Process Management

Process Management is primarily responsible for the creation, maintenance and destruction of Cisco IOS kernel processes, and/or tasks. The interface for yielding, or suspending the processor for other tasks is also part of Process Management.

A *task* in Cisco IOS is a unit of code that can be scheduled for execution independent of other units of code. The term “process” implies independent address spaces being assigned to each process and is not appropriate to use because Cisco IOS implements a shared memory architecture in which all Cisco IOS tasks share complete access to a common address space.

For more specifics on the Cisco IOS memory implementation, refer to the “Memory Management” chapter.

3.3.1 Cisco IOS Task Priorities

Ideally, the priority of a task controls how often a task gets processor (CPU) time. The scheduler uses multiple queues to track the tasks that run various priorities. At any given time, a task may occupy one and only one task queue.

The scheduler implements the following priority levels:

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

Caution A single poorly behaved task running at critical priority can disable a router because an item on this queue has the chance to execute after every high-, normal-, and low-priority task. Therefore, the use of this priority level *must* be approved by a senior engineer. Use the “interest-ios”, “kernel-trolls” or “interest-os” email aliases to contact a senior engineer.

- High. This priority is for tasks that require a faster than normal response to any of the events that they are waiting on, or for tasks whose work is important enough that the task should be scheduled ahead of the vast majority of other tasks in the system. High priority tasks require a lower than average *latency* (such as packet forwarding, either to external interfaces or other internal tasks).
- Normal. This is the default priority in which most tasks, 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.

3.3.2 Cisco IOS Task States

Note Process Management tracks more states than the scheduler because the scheduler does not need the level of detail Process Management tracks.

All Cisco IOS tasks exist in one of a finite set of states that describe their current activity. Table 3-1 describes these states.

Table 3-1 Cisco IOS Task States

Task State	Description
Running	Task is currently executing in the CPU.
Ready to run	(SCHEDSTATE_READY) Conditions for executing have been fulfilled and the task is in the appropriate ready queue, waiting to run. This implies that the task was previously waiting for an event or a timer, and that event or timer has occurred.
Waiting for event	Task is awaiting completion of an event before being ready to execute. This state is set via one of the functions <code>process_wait_for_event()</code> or <code>process_wait_for_event_timed()</code> .
Sleeping (absolute time)	Task is suspended until a specific clock time. This state is set when <code>process_sleep_until()</code> is issued.
Sleeping (interval)	Task is suspended until a specific time interval from the current time has elapsed. This state is set when <code>process_sleep_for()</code> is issued.

Table 3-1 Cisco IOS Task States (continued)

Task State	Description
Sleeping (periodic)	<p>Task is suspended until a regular time period has elapsed from the time the task was last awoken.</p> <p>This state is set when the function <code>process_sleep_periodic()</code> is issued.</p> <p>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 task was awoken, not the time the task finished executing.</p> <p>This allows a task to execute at a fixed interval and not have its wakeup time slowly drift because of processing time and time on the ready queue.</p> <p>For example, suppose a task should execute every minute, and it requires 1 second to perform its processing and the time on the ready queue is 5 seconds. Using the <code>process_sleep_periodic()</code> function, the tasks would execute at times 0, 1, 2, 3, 4, and so on. Using the <code>process_sleep_for()</code> function, which places a task in the “sleeping (interval)” state, this same task would execute at times 0:00, 1:06, 2:12, 3:18, and so on.</p>
Sleeping (managed timer)	<p>Task is suspended until a managed timer has expired.</p> <p>This state is set when the function <code>process_sleep_on_timer()</code> is issued.</p>
Hung	<p>Task would not relinquish the processor and was stopped by the watchdog interrupt. This task is never again scheduled to receive the processor.</p> <p>This state occurs only when a watchdog timeout has occurred and the resultant action of “hang” has been specified via scheduler process-watchdog hang.</p>
Dead	<p>Task has been killed and never resumes. This is a transient state that lasts until the scheduler can perform a postmortem analysis on the task and reclaim scheduler-allocated resources.</p>

3.3.3 Managing Cisco IOS Tasks

This section describes how IOS tasks are managed.

3.3.3.1 Overview

Once a task has been created and runs, it can perform the following operations:

- 1 Delay until system is initialized. Many tasks want to delay execution until the system is initialized and the configuration file processes because most tasks are created by initialization code. There are a few special functions to provide support for this.
- 2 Register for notification for the particular events for which it is to be awakened.
- 3 Wait for events and “sleep”. We refer to a “sleep” if waiting for a timer to expire; “waiting” if other events are to awaken the task. A waiting or sleeping task is placed on the idle queue until a timer or event causes them to be awoken.

- 4 Suspend processing, that is, give up the processor, but request to be placed on the ready list instead of the idle list. This is done if a task has more work to do before waiting for another event or timer.
- 5 Send a Direct Event to itself, or another task. This is not often used.
- 6 Determine the reason(s) a task is awakened, such as a timer expiration, change to any of a number of managed object types, direct waking from other code in the system.
- 7 Manipulate the set of managed objects and timers, by making requests to the scheduler for tasks such as:
 - Create a managed object or timer.
 - “Watch” that managed object/timer and notify your task when changed or expired.
 - Modify a managed object or timer.
 - Delete a managed object.
- 8 Set information about a task, such as the reasons to be awoken (perhaps masking off certain managed object types)
- 9 Query information about a task, such as the task priority, or the reasons to be awoken.
- 10 Terminate a task. This task may terminate itself, or another task.

3.3.3.2 Creating a Cisco IOS Task

New tasks can be created at initialization or at any time by any task. Tasks are generally created either as part of the startup process—from either the initialization process or a subsystem initialization routine—or they are created by a configuration command.

All Cisco IOS tasks are created using the `process_create()` function. The stack is allocated using `malloc()` and filled with `0xffffffff` for every `process_create()` call. The function itself does not pass any arguments to the new Cisco IOS task and does not provide the new Cisco IOS task with a controlling terminal.

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

The task creation function itself does not provide for initial arguments and does not have a terminal attached to it. If the task requires an argument or a terminal, this is the responsibility of the created task and is set by a call to one of the Cisco IOS task attribute modification functions after the Cisco IOS task has been created and before it executes.

It's not a good idea to declare large automatic variables when using `process_create()`, because they can overflow the IOS process stack (recursive algorithms can also overflow the process stack). In IOS, a process's stack is fixed in size and cannot grow. A stack frame larger than 1024 bytes will cause a static analysis warning.

When a Cisco IOS task is created it is placed in the idle state and then moved to the ready state. This transition moves the new task from the idle queue to the end of the appropriate ready queue. Once scheduled, the new task begins executing at the entry point supplied as an argument to `process_create()`. When `process_create()` is called by a Cisco IOS task, the calling Cisco IOS task does not yield the CPU as part of the Cisco IOS task creation process. The newly created Cisco IOS task will not be given a chance to run until the calling Cisco IOS task yields the CPU. If `process_create()` is called during kernel initialization, the new Cisco IOS task will not be given a chance to run until the scheduler begins executing.

Once a Cisco IOS task has been created, it is placed on the appropriate ready queue and the code segment issuing the `process_create()` continues. At this time, that code may 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.

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

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

For tasks 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.

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

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

3.3.3.2.1 Examples

The following example creates a task for the TCP discard daemon. The task is named `TCP Discard`. It has a process stack size of `NORMAL_STACK` and runs at low priority. After the task 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 task, passing an argument to the task and assigning the console as the output device:

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

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

3.3.3.3 Starting a Cisco IOS Task

As mentioned above, when a task is created it is placed on the ready queue with the `padd` routine as the routine to be executed by the task.

The `padd` routine performs setup such as allocation of memory for structures. If the task is to be awoken later based on changes in managed objects, it notifies Event Management of those managed objects during initialization.

When initialization is complete, the task can begin to execute.

3.3.3.4 Suspending a Cisco IOS Task

Tasks share the CPU with other tasks by requesting their willingness to suspend their operation. This is done through one of several different function calls that cause the scheduler to gain control. When this happens the scheduler simply stops executing the current task/process and begins executing the

next task/process in the priority queue. Otherwise, it follows the scheduling algorithm and does not execute the yielding task/process again until the scheduler's next iteration through that priority's ready queue.

A Cisco IOS task may suspend immediately. All tasks/processes are required to suspend/yield at some point, or a task/process may use a function that requests that the scheduler check if suspension is recommended and if so do the suspension. Finally, you may first perform a check and, if suspension is recommended, save state information and then issue an immediate suspend/yield.

There are 3 different calculations for determining when to suspend:

- The task has used up its “process quantum” or recommended time slice, which is usually 200 ms by default, or can be set using the `ios# process-max-time milliseconds` command.

The `process_time_exceeded()` function tests this condition and returns TRUE if the task has reached or exceeded its process quantum.

- Either the task has used up its process quantum (as indicated above) or there are one or more critical or high tasks ready to run.

The `process_would_suspend()` function tests for the conditional, and returns a TRUE or FALSE value, but does not suspend/yield.

The `process_may_suspend()` function tests for the conditional, and suspend/yields if the condition is TRUE.

- The task has reached the “maximum task time”, indicating that the task is hogging the CPU. The maximum task time is 2000 ms by default, or can be set using the `ios(config)# scheduler max-task-time milliseconds` command.

The `check_cphog()` function tests this condition and returns TRUE if the task reached or exceeded the maximum task time.

Table 3-2 lists the functions available for suspending a task.

Table 3-2 Functions for Checking and Suspending Tasks

Purpose	Effect	Function
Suspend immediately.	Place the task at the end of the appropriate ready queue. The task executes again during the scheduler's next pass of that queue.	<code>process_suspend()</code>
If the process has used up its process quantum, call <code>process_suspend()</code> .	Suspend this task if it has used its “process quantum”. The task is suspended as indicated above.	<code>process_suspend_on_quantum()</code>

Table 3-2 Functions for Checking and Suspending Tasks (continued)

Purpose	Effect	Function
Conditionally relinquish the processor.	<p>Suspend this task if there are one or more higher priority tasks ready to run.</p> <p>The task is suspended as indicated above.</p> <p>This function does allow scheduler support for guaranteed quantum for Normal/Low priorities.</p> <p>The <code>process_may_suspend()</code> function suspends a low-priority or normal-priority process when a high-priority or critical-priority process is ready to run. In this case, the low-priority or normal-priority process is rescheduled after the high-priority or critical-priority process releases the CPU. The rescheduled low-priority or normal-priority process will then receive the remainder of its CPU quantum.</p> <p>Calling the <code>process_may_suspend()</code> function frequently reduces the scheduling latency for high-priority or critical-priority processes. However, this ensures that a low-priority or normal-priority process will receive its full CPU quantum, even when temporarily suspended as a result of running a high-priority or critical-priority process.</p>	<code>process_may_suspend()</code>

Table 3-2 Functions for Checking and Suspending Tasks (continued)

Purpose	Effect	Function
If the process has used up its process quantum, call <code>process_may_suspend()</code> .	<p>Suspend this task if it has used its “process quantum” and if there are one or more higher priority tasks ready to run. If both conditions are true, the task is suspended as indicated above.</p> <p>The use of this function is strongly discouraged. All processes should use <code>process_may_suspend()</code> or <code>process_would_suspend()</code> instead. These recommended alternatives decrease the scheduling latency for high-priority and critical-priority processes without negatively impacting your process. Your process continues to receive its full CPU quantum, even when temporarily suspended as a result of running a high-priority or critical-priority process. The difference between <code>process_may_suspend_on_quantum()</code> and <code>process_may_suspend()</code> or <code>process_would_suspend()</code> is that <code>process_may_suspend()</code> immediately releases the CPU to a high-priority or critical-priority process that is runnable. Once that process completes, the process that was suspended by <code>process_may_suspend()</code> gets rescheduled for the remainder of its quantum. <code>process_would_suspend()</code> is hypothetical: it shows you what the result would be if you called <code>process_may_suspend()</code> but does not actually suspend the process. <code>process_may_suspend_on_quantum()</code>, on the other hand, does not release the CPU at all until the process has used the specified percentage, regardless of whether any high-priority or critical-priority process is runnable.</p>	<code>process_may_suspend_on_quantum()</code>

Table 3-2 Functions for Checking and Suspending Tasks (continued)

Purpose	Effect	Function
	<p>Suspend this task if it has used its allocated portion of CPU time; task is suspended as indicated above.</p> <p>The use of this function is strongly discouraged. All processes should use <code>process_may_suspend()</code> or <code>process_would_suspend()</code> instead. These recommended alternatives decrease the scheduling latency for high-priority and critical-priority processes without negatively impacting your process. Your process continues to receive its full CPU quantum, even when temporarily suspended as a result of running a high-priority or critical-priority process. The difference between <code>process_may_suspend_on_quantum()</code> and <code>process_may_suspend()</code> or <code>process_would_suspend()</code> is that <code>process_may_suspend()</code> immediately releases the CPU to a high-priority or critical-priority process that is runnable. Once that process completes, the process that was suspended by <code>process_may_suspend()</code> gets rescheduled for the remainder of its quantum. <code>process_would_suspend()</code> is hypothetical: it shows you what the result would be if you called <code>process_may_suspend()</code> but does not actually suspend the process. <code>process_may_suspend_on_quantum()</code>, on the other hand, does not release the CPU at all until the process has used the specified percentage, regardless of whether any high-priority or critical-priority process is runnable.</p> <p>high or critical priority process running.</p>	<code>process_suspend_if_req()</code>

Table 3-2 Functions for Checking and Suspending Tasks (continued)

Purpose	Effect	Function
Check whether a task has exceeded the allotted “process quantum”.	The use of this function is strongly discouraged. All processes should use <code>process_may_suspend()</code> or <code>process_would_suspend()</code> instead. These recommended alternatives decrease the scheduling latency for high-priority and critical-priority processes without negatively impacting your process. Your process continues to receive its full CPU quantum, even when temporarily suspended as a result of running a high-priority or critical-priority process. The difference between <code>process_may_suspend_on_quantum()</code> and <code>process_may_suspend()</code> or <code>process_would_suspend()</code> is that <code>process_may_suspend()</code> immediately releases the CPU to a high-priority or critical-priority process that is runnable. Once that process completes, the process that was suspended by <code>process_may_suspend()</code> gets rescheduled for the remainder of its quantum. <code>process_would_suspend()</code> is hypothetical: it shows you what the result would be if you called <code>process_may_suspend()</code> but does not actually suspend the process. <code>process_may_suspend_on_quantum()</code> , on the other hand, does not release the CPU at all until the process has used the specified percentage, regardless of whether any high-priority or critical-priority process is runnable.	<code>process_time_exceeded()</code>
Check whether there are one or more critical or high tasks ready to run or a task has exceeded the allotted “process quantum”.	—	<code>process_would_suspend()</code>
Check whether a task has reached the maximum task time, which indicates a CPU hog condition.	—	<code>check_cphog()</code>
Determine whether a task can suspend. A task can suspend if it is neither running at interrupt level nor has blocking been disabled.	—	<code>process_suspend_allowed()</code>

Table 3-2 Functions for Checking and Suspending Tasks (continued)

Purpose	Effect	Function
Safely suspend unconditionally.	Place the task at the end of the appropriate ready queue. The task executes again during the scheduler's next pass of that queue.	process_safe_suspend()

3.3.3.5 Setting and Retrieving Information about a Cisco IOS Task

Table 3-3 lists the functions available to set and retrieve information about tasks.

Table 3-3 Set and Retrieve Information about a Task

Information about Task	Function to Set	Function to Retrieve
Name	Parameter to process_create() process_set_name()	process_get_name()
Identifier (PID)	Returned by process_create()	process_get_pid().(ios_getpid()) also returns the PID for the currently executing process. This function is quicker and does not take parameters like process_get_pid()).
Priority	Provided to process_create() process_set_priority()	process_get_priority()
Controlling terminal	process_set_ttynum() process_set_ttysoc()	process_get_ttynum() process_get_ttysoc()
Socket structure	process_set_socket()	process_get_socket()
Profiles	process_set_profile() process_set_all_profiles()	process_get_profile()
Stack size	Provided to process_create()	process_get_stacksize()
Classes of events allowed to wake up a task	process_set_wakeup_reasons()	process_get_wakeup_reasons()
Time task started this execution	—	process_get_starttime()
Cumulative running time in milliseconds since task created	—	process_get_runtime()
Arguments that are passed to task at time of creation	process_set_arg_num() process_set_arg_ptr()	process_get_arg_num() process_get_arg_ptr()
Whether to perform postmortem stack analysis of process	process_set_analyze()	process_get_analyze()
Whether task should stop running while a core dump is being written	process_set_crashblock()	process_get_crashblock()
For testing, indicate debugging required for this task	process_set_debug()	--
For testing, modify start time and number of times this task has executed	process_set_timing()	--

3.3.3.5.1 Process_get_analyze() and process_set_analyze() Guidelines

`process_set_analyze()` manages the field in the `sprocess` structure
`some_process->analyze`.

This field is only used to determine “stack analysis” when a task is killed. Because “stack analysis” has minimal overhead in instructions and memory and may be valuable in recording dead tasks, it is recommended that this always be set to TRUE.

When `some_process->analyze = TRUE`, the following occurs:

- Calculate “available stack bytes at death” for this process’ stack.
- If not already existing,
 - Allocate a new post-death stack-header structure (`stacktype`).
 - Initialize stack-header fields:`process_name`; total bytes in stack; “available stack bytes at death.”
 - Enqueue on list of all post-death stack header.
- If already exists (task of this name has died before)
 - Compare previous “available stack bytes at death” with new calculation.
 - If new number is less than previous, save new value in stack-header structure.

3.3.3.6 Determining Whether a Cisco IOS Task Exists

Two functions—`process_exists()` and `process_is_ok()`—allow you to determine whether a task exists. To determine whether a process identifier (PID) exists use the function

```
boolean process_exists(pid_t pid);
```

To check that a task exists and also check that it has not failed, use

```
boolean process_is_ok(pid_t pid);
```

3.3.3.7 Destroying a Cisco IOS Task

A Cisco IOS task can be destroyed by killing itself, by having another Cisco IOS task kill it, or by having the scheduler kill it because it has grossly overused the processor and is assumed to be in an endless loop.

When a Cisco IOS task kills itself, the main routine 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 Cisco IOS tasks are no longer needed—for example, when a protocol is unconfigured—they should clean up after themselves and kill themselves.

Upon death, the Cisco IOS task’s `SIGEXIT` signal routine is driven and the Cisco IOS task is placed on the Dead queue. This `SIGEXIT` routine performs cleanup such as disabling and deleting managed objects and timers and returning memory acquired by the Cisco IOS task.

Later, as part of the cleanup routine of the scheduler loop, the following occurs: internal scheduler memory, such as the process structure, the stack, and memory to hold scheduler events, is returned; the unique PID number is recovered and made available for future Cisco IOS tasks.

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

```
void process_kill(pid_t pid);
```

If a SIGEXIT signal handler was specified (via `signal_permanent()` function), that routine is called before the task is destroyed by return of the PID and scheduler memory for that task.

3.3.4 Example

The following example shows code for creating and exiting from a Banyan VINES router Cisco IOS task. Creating and exiting from a task is handled by Process Management. The creation of managed objects is handled by Event Management.

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);
    .
    .
    .
}
```

Exit Handler

Exiting and cleaning-up the task is handled via Process Management and Event Management.

```
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;
}
```

3.3.5 Process Accounting and CPU Utilization

This section describes how Process Management records CPU utilization and manages process accounting, and how to implement more accurate accounting of CPU usage.

3.3.5.1 Process Accounting Threshold Values

Table 3-4 summarizes the threshold values for accounting purposes and process management (see the functions described in Table 3-2). The scheduler uses the watchdog to clean up after a process that appears to be hung.

Table 3-4 Process Accounting Thresholds

Value	Description	How to Set	Default Assignment
Process Quantum	The maximum desirable process time slice in ms.	process-max-time command sets this value.	200ms (most platforms)
Maximum Task Time	The maximum time a task can run before signaling a CPUHOG condition.	scheduler max-task-time command sets this value.	2000 ms (2 s) 20 ms for preemptive processes. See Section 3.3.6.1, “Flow of Preemptive Processes”.
Watchdog Timeout	The interval after which the scheduler assumes a process has hung and needs to be killed (see Table 3-1, “Cisco IOS Task States”).	scheduler process-watchdog command sets the watchdog timeout action.	2 minutes

3.3.5.2 CPU Usage Accounting

CPU usage of a Cisco IOS task is measured by a non-maskable timer interrupt service routine (ISR) that occurs every 4ms. The timer ISR checks the current context, which will be one of the following:

- Process-level
- Interrupt-level
- In the scheduler

The current context is charged with utilizing the CPU for that 4ms.

3.3.5.3 Fixing CPU Usage Accounting Inaccuracies

The timer ISR checks the current context in an attempt to avoid holding processes accountable for CPU time spent servicing interrupts during the process time slice, and vice versa for interrupt-level code accountability. However, at each timer tick, the timer ISR applies the full 4ms of CPU time to the current context. As a result, depending on when the timer tick occurs in relation to a context switch, you might see inaccuracies in CPU utilization accounting compared with the actual computation time because some or all of the tick is being applied to the wrong context.

For example, if a process-level task is scheduled just after the timer tick, but suspends before the next timer tick, that task has none of that CPU utilization time charged to it. The context switch happens before the timer ticks, the timer ISR interrupts in the new context, and credits the new context with the entire previous 4ms of CPU time. Similarly, if a process is interrupted for most of a tick interval but regains the processor before the timer ticks, the entire 4ms is counted towards the process utilization time although most of that time was actually spent at interrupt level.

Process context switches and timer ticks might align such that the CPU utilization count for a process is out of proportion to its actual usage, which appears as CPU usage “spikes” in the output of the **show process cpu** command. This condition can affect fair scheduling and response times of all processes.

(Available in 12.2S and above for MIPS 4K and related platforms:) A fix for this problem is available in which the timer ISR, the scheduler, and platform-specific ISRs more accurately record the time spent in each context, and take the differences into account when recording CPU utilization.

Note See CSCec17882 and the software design document, EDCS-314181, for more details on this enhancement.

Enable it on MIPS 4K and related platforms as described in the steps below.

Use the following code as an example for porting this enhancement to other platforms:

“sys/asm-mips/asm_4k.h” for the `SAVE_INTR_START_TIME` and `UPDATE_INTR_CPU_USAGE` macros, and “sys/asm-mips/asm_4k_timer.S” for the timer ISR.

If you observe that process accounting inaccuracies are unfavorably affecting process scheduling, you can enable this enhancement as follows:

- Step 1** Call the `enable_correct_accounting()` API function at initialization time to enable more accurate accounting of time spent in each context.
- Step 2** Add a call to the `SAVE_INTR_START_TIME(reg1, reg2, reg3, reg4)` assembly language macro at the beginning of your platform-specific ISRs to store the ISR start time when entering interrupt context.
- Step 3** Add a call to the `UPDATE_INTR_CPU_USAGE(reg1, reg2, reg3, reg4)` assembly language macro at the end of your platform-specific ISRs to compute the time spent in the ISR.

This process accounting enhancement also provides a mechanism to detect that an ISR is hung and handle a watchdog timeout on an ISR. You can enable or disable this feature using the **[no] scheduler isr-watchdog** command.

If not already available on your platform, in addition to implementing the ISR accounting macros, you must also add code to your platform’s 4ms timer ISR to record the time spent at interrupt level. As an example, the following pseudocode illustrates how the MIPS 4K timer ISR implements this feature:

```

/*
 * The following code is added for correct interrupt accounting.
 * If the platform supports correct accounting then it needs
 * to set intr_acc_enabled global to TRUE. Setting it so,
 * will make the accounting happen with respect to the interrupt
 * usage.
 *
 * if(int_acc_enabled) {
 *
 *   if(r4k_raise_intr_count) {
 *     int_count++
 *     shadow_int_count++
 *     goto idle_out
 *   }
 *
 *   if(r4k_int_cpu_usage_ticks > r4k_4ms_pipe_ticks) {
 *     r4k_int_cpu_usage_ticks -= r4k_4ms_pipe_ticks
 *     int_count++
 */

```

```

*      }
*
*
*      if(int_busycount) {
*          if(getcp0ticks - r4k_int_time_start >= r4k_4ms_pipe_ticks){
*              int_count++
*              r4k_int_time_start += r4k_4ms_pipe_ticks
*          }
*      }
*      check_watchdog()
*      goto handle_watchdog
*  }
*/

```

3.3.6 How to Preempt IOS Processes

With the current design of IOS, processes run with a cooperative multitasking, run to completion model. Before process pseudo-preemption was provided in IOS, if you needed to do some work at process level urgently, it was not possible to preempt the running process and schedule any other process. The urgent process had to wait for the currently running process to yield the CPU. However, some situations demand urgent processing at the process level, for example BFD (Bi-directional Forwarding Detection). Thus, process pseudo-preemption in IOS was first designed to accommodate BFD in 12.2S and was later ported to the MIPS platforms in 12.0S per the “Commits-note” enclosure of CSCCec04080. It was designed to provide the infrastructure that is required to preempt IOS processes to do some extremely rare and critical processing at the process level, such as that needed by applications like BFD (for information on BFD, see EDCS-379295 and EDCS-381561).

Note The creation of preemptive processes will be policed and must be justified with a strong case and with profiled data.

This subsection is based on EDCS-250086. The following topics are presented in this subsection:

- Flow of Preemptive Processes
- Changes to the sprocess Structure for the Pseudo-preemption Infrastructure
- Pseudo-preemption Infrastructure Considerations and Restrictions
- Preemptive Process Capabilities
- API for the IOS Pseudo-preemption Infrastructure

3.3.6.1 Flow of Preemptive Processes

Process pseudo-preemption in IOS provides a new class of “preemptive processes” and a new queue for them. When preemptive processes are created, these processes move to ready queue, as usual. The flow of preemptive processes continues as follows:

- 1 Preemptive processes have a quantum of 20mS and they may block. If a preemptive process exceeds this quantum, an error message is displayed, but the process is allowed to run to completion. The CPUHOG will fire at crossing 20mS; however, the watchdog threshold of two minutes is not changed.
- 2 Preemptive processes may communicate with other classic processes via events. An event may be posted by an ISR (Interrupt Service Routine) or a process. When an event is posted, the receiving process, as usual, is moved from the idle queue to the ready queue.

- 3 If an event is posted for a preemptive process from process-level code with no interrupts disabled, then the preemptive process(es) will be run suspending the current process immediately. If it is posted from an ISR (or) from process-level code that has disabled interrupts, a software interrupt set at the lowest priority will be scheduled that will run when it is safe to run (as soon as interrupts are enabled). The software interrupt that is scheduled must have the lowest priority in the interrupt table. This ISR modifies the stack of the running process to jump to the preemption handler. Thus, if a classic process has disabled interrupts to protect against shared data, then it is not preempted.
- 4 At the same time, since the classic process does not anticipate any other process to run and corrupt its data, the preemptive process cannot touch any shared data. However, if the preemptive process needs to touch any shared data, it is the responsibility of the application to protect this data.
- 5 Software interrupt is specific to the MIPS processor, so other processors may not support it. For example, PPC (PowerPC) Motorola processors do not. For those that do not have a MIPS processor, a check is done to see if preemption is needed from the interrupt unwind code. If so, the return address to the preemption handler is changed as in MIPS case. Note that care should have been taken to make sure to avoid being on a nested ISR stack.
- 6 The process preemption handler runs completely at the process level... saves the context of the preempted process, and forces a suspend on the current process.
- 7 The scheduler takes over hereafter, and it was modified so that it runs each process from the preemptive ready queue.
- 8 These new processes need to be quick and should not touch shared data. But they are allowed to block. If they must touch the shared data, the shared data must be coded (by the application programmer) to protect its data from other processes (this is similar to the way a process protects its shared data from ISR). The API functions that were introduced for this purpose are `disable_preemption()` and `enable_preemption()`. For details on these API functions, see subsection 3.3.6.5, “API for the IOS Pseudo-preemption Infrastructure”.
- 9 These preemptive processes are allowed to do `malloc()` operations. In the event that the preemptive process blocks, the control shall return to the scheduler that schedules back the preempted process.
- 10 To support `malloc()`, a semaphore is used. If pseudo-preemption occurs when a classic process is in the middle of a `malloc` operation, the preemptive process blocks on the semaphore, and thus returns control back to the preempted classic process. The classic process goes ahead and finishes the `malloc()` operation and unlocks the semaphore, thus waking up preemptive processing, which preempts the classic process again. With the critical sections in `malloc` library spread over many APIs, this semaphore is “recursive”, that is, consecutive locks by the same process will succeed (for more information, see the `create_watched_recursive_semaphore()` API reference page).
- 11 Current `malloc()` support is using a semaphore. Semaphore code calls `malloc()` in turn for blocking the caller (one who does not succeed in locking). This problem is solved by creating `sched_event_set()` and `watcher_info()` on the stack of the caller instead of doing a `malloc`.
- 12 The preemptive processes are also allowed to call the basic `process_xxx()` APIs and the event management APIs, such as the `create_watched_xxx()` APIs. In order to support this, the critical sections in all these relevant APIs must be protected from preemption using the `disable_preemption()` and `enable_preemption()` APIs.
- 13 To allow `process_create_common()` to be called from preemptive processes, it was made reentrant, by using a semaphore.

- 14** In the case that the preemptive process cannot finish in its quantum, a CPUHOG is reported from the timer ISR, but allows the preemptive process to run to completion. If the preemptive process tries to suspend, as opposed to going to the idle queue, it is not allowed to suspend and it continues doing its work until it exceeds its quantum, at which point it is flagged as a CPUHOG process to catch buggy code.

A preemptive process trying to suspend prematurely is not an error condition and so it is not flagged for trying to suspend. The rationale behind not allowing it to suspend is same as the rationale for critical and high priority processes, which are not supposed to stay in the ready queue for a long time; instead, they should finish their work as soon as possible and go to the idle queue.

Only highly critical and quick jobs should be done from a preemptive process, as opposed to doing lot of work. If a lot of work needs to be done, it should not be a preemptive process in the first place or it should be redesigned to be split and work with a “companion process”. Also, profiling the preemptive process should be done to avoid CPUHOG

- 15** Next, the process from preemptive queue, if any, starts running with a quantum of 20mS.
- 16** When one round of all processes in preemptive ready queue are finished, the scheduler reruns the process that was forced to suspend. At this point, the preemptive handler is returned and the preempted process context is restored as if nothing has happened. The preempted process can continue as before and when it suspends itself, the scheduler takes charge.
- 17** Blocking disabled-processes are also preempted by the preemptive process. Processes can disable blocking by calling the `set_blocking_disabled()` API. This flag allows modules to detect when the functions that they in turn call are suspending the process when they should not be. Blocking means any kind of descheduling point, to run other processes. But preemptive processes are not included in this, and they run preempting a blocking disabled-process.

3.3.6.1.1 Preemptive Processes and the Scheduler

The scheduler cannot be preempted. If an event is posted for a preemptive class of process while scheduler is running, the preemption is serviced as soon as the scheduler runs any classic process from the ready queue. The preemptive process queue is also serviced like other queues. The position of this servicing is inside critical queue servicing.

3.3.6.1.2 Preemptive Processes and Timer Events

Timer events are checked by scheduler when a process suspends. Thus, a preemptive process cannot be woken up in pseudo-preemptive fashion for timer events without taking a substantial performance hit in timer ISR. Hence, for this purpose it is recommended to use timer wheels instead.

Note Timer events are not “posted” by process or an ISR.

3.3.6.2 Changes to the sprocess Structure for the Pseudo-preemption Infrastructure

For troubleshooting and debugging purposes, a pointer to a preemption context structure is maintained in the `sprocess` structure. The pointer has information on which preemptive process(es) preempted this normal process and at what point, that is, the PC(s) at the time of preemption.

For accounting purposes, the following three fields were added to the `sprocess` structure:

- 1 The field used to maintain the amount of time spent on running preemptive processes when this process was running, in one schedule.
- 2 The field used to maintain the amount of time spent on interrupts when running preemptive processes, while this process was running, again in one schedule.
- 3 The field for the timestamp used to maintain watchdog calculations. This is separate from the field used to maintain the amount of time spent running preemptive processes when the process was running (1) because the watchdog would not fire, if the process was friendly enough by calling `process_may_suspend()` often.

3.3.6.3 Pseudo-preemption Infrastructure Considerations and Restrictions

You should keep in mind the following regarding IOS Pseudo-preemption Infrastructure:

- Pseudo-preemption Infrastructure Memory and Performance Considerations
- Pseudo-preemption Infrastructure Configuration Considerations
- Pseudo-preemption Infrastructure Restrictions

3.3.6.3.1 Pseudo-preemption Infrastructure Memory and Performance Considerations

Memory and performance are impacted by the pseudo-preemption infrastructure as follows:

- 1 The context of a preempted process is saved on the stack of the preempted process, and hence there is no need to allocate memory in `sprocess`. However, three `sprocess` fields were added for accounting purposes, and a circular array of size 10 to hold the last 10 processes that preempted the process and the points of preemption for troubleshooting purposes. This might have some memory impact.
- 2 If the preemptive process needs to touch shared data, then it needs to be protected by disabling interrupts, by using some other mechanism (such as a lock or semaphore), or by using the `disable_preemption()` and `enable_preemption()` functions. This might cause some performance impact.
- 3 The critical sections within the infrastructure are protected with the `disable_preemption()` and `enable_preemption()` pair of API functions; this might cause some delay for preemptive processes.
- 4 To keep `malloc()` and `free()` safe from preemptive processes, all the critical sections of the `malloc` library that might get disturbed by preemptive processes executing `malloc()` or `free()` are protected with a semaphore specific to each mempool (see the `create_watched_recursive_semaphore()` API reference page). Locking and unlocking can cause some performance impact, but this happens only when at least one preemptive process is created.
A preemptive process takes control of the scheduler, while a classic IOS process that holds a semaphore is in the middle of a `malloc()` or `free()`. At the start of a `malloc()` or `free()`, all processes try to acquire the semaphore. However, as the classic IOS process holds the semaphore, the preemptive process goes into an idle state. The normal IOS process completes the `malloc()` or `free()` and releases the semaphore. The preemptive process wakes up and takes hold of the semaphore, and completes the `malloc()` or `free()`.
- 5 Preemption is not allowed while in the scheduler, so there is a check to see if there are any events pending for preemptive processes, just before the context switch from scheduler to process. This can cause some minor performance impact. Minor performance impact is also possible when the API notifies a process/processes of an event occurrence.

- 6 For MIPS-based platforms, if they use software interrupts for any other purpose, then they will be multiplexed with Cisco IOS usage for preemption, which might cause some minor delay in servicing them.
- 7 For PPC-based platforms, ISR-level preemption is checked from the interrupt unwind code. This can have a little performance impact.

3.3.6.3.2 Pseudo-preemption Infrastructure Configuration Considerations

No configuration is required for the IOS Pseudo-preemption Infrastructure.

3.3.6.3.3 Pseudo-preemption Infrastructure Restrictions

The restrictions on the preemptive processes are as follows:

- 1 The preemptive processes should be ISR-like, fast, and not touching shared data. However, the ISR-unlike preemptive processes can be blocking (for instance, `process_wait_for_event()` or `process_suspend()` or such relevant calls that block the process for an event can be called from preemptive processes). They should not call any functions (or chains) that will touch the shared data. If a preemptive process needs to touch shared data, it is the responsibility of the application to ensure a protection. All `process_xxx()` calls and `malloc()` will be supported.
- 2 A preemptive process must not preempt another preemptive process; in other words, it must not post an event to another preemptive process.
- 3 The scheduler cannot be preempted.
- 4 Preemptive processes can run only when interrupts are enabled. In other words, the preemptive process is not guaranteed to run in 50mS; however, it will run asap (as soon as the interrupts are enabled).

Note The infrastructure or application code may have the preemption disabled while running its critical sections.

- 5 All MIPS platform code *must* adjust the interrupt priorities such that software interrupt has the least priority, it must only execute from a process level and cannot be nested with other ISRs. (This can be done for example for MIPS R4K family with the help of `r4k_level_table` or the mask in status register.) If the software interrupt is already used, platform software ISR code needs to be modified so that it simulates polling, which allows multiplexing of different handlers for the same level as the software interrupt.
- 6 A preemptive process cannot be woken up in pseudo-preemptive fashion for timer events.

3.3.6.4 Preemptive Process Capabilities

A preemptive process created by `process_create_preemptive()` can do the following:

- 1 It can do whatever an ISR can do.
- 2 It can allocate and deallocate mempool memory using `mempool_malloc()` and the corresponding `free()` calls.
- 3 It can call `process_create()`.

- 4 It can perform the basic event management functions that a process would need to do (that is, the `create_watched_xxx()`, `process_watch_xxx()`, `process_wait_for_event()`, `process_get/set_xxx()` calls, and the relevant API functions for triggering an event or for getting the value of the managed object).
- 5 It can kill itself via `process_kill()`.

Note Preemptive processes are allowed to run to completion, and they cannot suspend.

Note Preemption happens only when interrupts are enabled; a preemptive process is not guaranteed a run within 50ms.

3.3.6.5 API for the IOS Pseudo-preemption Infrastructure

The following IOS API functions are provided to support infrastructure (see the API reference pages for detailed information on these functions):

- 1 `create_watched_recursive_semaphore()`
- 2 `disable_preemption()`
- 3 `enable_preemption()`
- 4 `process_create_preemptive()`

3.3.6.5.1 Create a Preemptive Process

To create a preemptive process, call the `process_create_preemptive()` function:

```
pid_t process_create_preemptive(process_t *,
                                const char *,
                                stack_size_t,
                                process_priority_t,
                                preempt_id_t);
```

The arguments for this function are similar to `process_create()`, with the exception of `preempt_id_t`, which is an identifier that is maintained by infrastructure. This identifier is used to control the creation of preemptive processes. Applications need to present profiling data before the processes are allowed to be created as preemptive processes. The `process_create_preemptive()` function returns the PID of the process created on success or returns `NO_PROCESS` on failure.

3.3.6.5.2 Disable Preemption

To disable preemption in order to protect critical sections of caller context with any of the pseudo-preemptive processes, call the `disable_preemption()` function:

```
boolean disable_preemption(void)
```

This API can be used by applications to disable the preemption until the preemption is enabled through `enable_preemption()` API. The `disable_preemption()` function returns whether the preemption was disabled or not, before disabling it.

Note Applications should consider resolving such critical section problems with a semaphore before trying to use this API. Semaphores should cover most of the cases.

3.3.6.5.3 Enable Preemption

To enable preemption, call the `enable_preemption()` function:

```
void enable_preemption(boolean)
```

This API can be used by applications to enable the preemption as dictated by the Boolean argument that is returned by the `disable_preemption()` API. Typically, the return value from `disable_preemption()` is given as argument to `enable_preemption()` to make it back to the state when the `disable_preemption()` function was last called.

3.3.6.5.4 Create a Recursive Semaphore

The `create_watched_recursive_semaphore()` function extends the `create_watched_semaphore()` function to allow the current owner to acquire the semaphore multiple times. The `create_watched_recursive_semaphore()` function provides general support for recursive semaphores, which are semaphores that allow the owner to acquire the lock multiple times without deadlocking:

```
watched_semaphore *create_watched_recursive_semaphore(char *name, ulong id)
```

This API will create a recursive semaphore, which allows consecutive locks by the same process to succeed. There must be an equal number of unlocks for the semaphore to be released.

The preemption infrastructure required support for recursive semaphores, so it was added through this feature; however, other applications/infrastructures can use this in the future. When this feature is evaluated for porting to other platforms, plan for some effort to analyze the platform interrupt table/vectors/routines before enabling the pseudo-preemption infrastructure on the platform.

On MIPS-based platforms, we achieve preemption using the software interrupt (`sw1`) mechanism. If the platform was already using this level of interrupt for its own purposes, then this infrastructure's usage needs to be multiplexed with it. On the other hand, if the platform was not using software interrupt, and if the platforms had different ways of disabling/enabling interrupts (apart from the well-defined interfaces), then those need to be studied to make sure `sw1` is correctly included at the places where the interrupts are enabled/disabled.

On PPC based platforms, the changes are basically to the core platform interrupt dispatch routine, and the core routines that set/reset the interrupt level.

3.4 Event Management

Event Management, or Synchronization Management, is responsible for coordinating processes, tasks, and external events. Examples of system events include timer events that are used to wake sleeping tasks or I/O device events that are used to signal that a device is available or that its ready-status has changed. In still other cases, events provide a mechanism to synchronize the use of a real shared resource, like shared memory, or an abstract resource like a queue or table.

Events may be managed either individually or a collection of events may be managed as a set.

3.4.1 Managing Sets of Events

The scheduler provides three functions to manage sets of events:

- `process_push_event_list()` performs two operations; saves the task's currently active event set on the event stack by performing a LIFO push, and installs and activates set of events specified by the `sched_event_set` pointer. If the `sched_event_set` pointer is null, a null set of new events is installed.
- The newly created event set remains active until it is deactivated and removed by `process_pop_event_list()`.
- `process_pop_event_list()` performs two operations. The function replaces currently active event set by performing a LIFO pop of the IOS task's event stack. If no event sets are present, the function returns indicating the failed status of the operation. Also, if a storage pointer is passed, the function returns a pointer to the old event set, rather than releasing the memory.
- `process_get_wakeup()` tests the task's currently active event set to determine if any event has occurred.

3.4.1.1 How Event Sets Are Used

The purpose of the event-set functions is to improve the efficiency of Event Management through wholesale manipulation of the event environment.

For example, in the context of a given library, `process_push_event_list()` allows the library code to focus on the events associated with the library's function while simultaneously ignoring those of the caller. Once it completes its function, `process_pop_event_list()` allows it to restore the event-set of its caller.

A given library or service often depends on a fixed set of events to perform its function. When this is true and the routine is non-reentrant, it is possible to construct the set of events only once and to reuse it each time the library or service is called. Doing so requires the use of a persistent pointer to the prebuilt `sched_event_set`.

3.4.2 Example

The following example shows the event-set management code being used to manage an IP socket read for UDP:

```
sys/ip/udp.c: read_ipsocket

[...]
events_pushed = process_push_event_list(NULL);
process_watch_queue(soc->inq, ENABLE, ONE_SHOT);
process_watch_boolean(soc->errflag, ENABLE, ONE_SHOT);
process_wait_for_event_timed(time);
if (events_pushed) {
    /*
     * Disable and pop only if the push was successful
     * in creating a clean slate.
     */
    process_watch_queue(soc->inq, DISABLE, ONE_SHOT);
    process_watch_boolean(soc->errflag, DISABLE, ONE_SHOT);
    process_pop_event_list(NULL);
}
[...]
```

3.4.3 Managing Individual Events

This section describes how individual events are managed.

3.4.3.1 Event Types

Event Management provides primitives for managing the following types of events:

- Queues—An ordered collection of items that can be placed on the queue by one code segment and retrieved one time at a time by a task.
- Booleans—A Boolean that when changed from FALSE to TRUE, all the task(s) watching that Boolean are scheduled to execute.
- Bit fields—A 32-bit bit field. When the bit field is modified the task(s) watching that bit field are scheduled to run.
- Semaphores—A memory location used by multiple tasks to serialize access to a set of resources. When one code segment releases the semaphore, all the IOS tasks waiting are scheduled to attempt to gain the lock. The shared resource can only service a certain number of requests at a time.

The resource can be anything, for example, flash memory or the table of IP routes. To guarantee the atomicity of operations that modify these resources, you must lock the semaphore before making the modification and unlock the semaphore when the modification is complete.

Note Atomic locks are also available. An atomic lock 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. Atomic locks do not provide the scheduling function but simply return success or failure of the lock request. The atomic lock functionality does not provide the ability to block until a lock is released. That implies coordination with the scheduler that a managed semaphore provides.

- Read/Write locks, as the name implies, allow multiple execution units to share the read operation of data, whereas only one execution unit will be allowed to modify the protected data at any given instant of time.
- messages and watched messages—A message is a simple communications mechanism passing only an ID, numeric and a pointer. A standard message is always delivered immediately; watched messages can be disabled by ID.
- timers

Note Within a given category, each event is identified by a major and minor type code. Together, these form a unique identifier for the event “within its category”. If an event can be described fully by its major type code, the minor type code is specified as zero.

3.4.3.2 Event Operations

Event Management provides the following operations for managed objects:

- Creating and destroying an event. Each event is composed of a major event type, minor event type, and concerned IOS task. The major event type describes the type of managed object being waited upon, for example QUEUE_EVENT. The minor event type is a user-defined identifier that is associated per instance of a managed object.

- Registering for event notification. This is referred to as “watching” an event. Before going idle, tasks must notify the scheduler of the events for which they are to be awoken. Tasks can later disable the notification.

More than one task can watch a particular managed object, however this is unusual. Usually a single task watches a managed object. The most typical managed objects to have multiple watching tasks are Booleans; others are rare.

Explicit registration for notification is not required with messages and semaphores.

- Waiting for events. After indicating what managed object events to register for, an IOS task waits until another code segment causes an event to be generated for the managed objects it is registered with. When the event occurs, the IOS task is then scheduled to run. Waiting may be either timed or untimed. The primitives used to invoke waiting are:

```
void process_wait_for_event(void);  
void process_wait_for_event_timed(void);
```

- Generating an event. The following actions will generate an event that in turn will cause one or more tasks to be scheduled:

- Change of a managed Boolean from FALSE to TRUE.
- Enqueue of an item to a managed queue that is currently being watched.
- Modification of managed bit field, that has the *send_wakeup* flag set.
- Unlocking a managed semaphore, that is being watched by one or more tasks.

Some modifications or actions to managed objects do not result in the scheduling of watching tasks:

- Setting a managed Boolean when the set operation does not result in a transition of the Boolean from FALSE to TRUE (for instance, TRUE to TRUE, TRUE to FALSE, FALSE to FALSE).
- Removing an item from a managed queue; peeking at a queue does not alter it so the item does not belong on the list of scheduling changes.
- Querying queue size characteristics of a queue does not alter the queue, so the queue does not belong on the list of scheduling changes.
- Adjusting the maximum size of a queue.
- Modification of a bit field, with the *send_wakeup* flag unset.
- Locking a managed semaphore.
- Scheduling of one or all IOS tasks. The scheduler keeps a list of all tasks watching a particular event. On an object change, all the tasks watching the particular event are scheduled to execute, in the order of their registrations, i.e, the order in which the processes called `process_watch_xxx()`. If the task had set the `wake_first_only` flag, then the first task on the wakeup list is scheduled to execute and the event structure corresponding to this wakeup is moved to the end of the wakeup list. By this, when the event occurs again, the next task in the event’s wakeup list is scheduled to execute. This results in a round robin execution of tasks as object changes occur

Note This is valid only if the task had set the `wake_first_only` flag for this event.

3.4.3.3 Queues: Definitions, Classes, and Attributes

This section describes the characteristics of managed queues. In this section, the terms “managed” and “watched” are used interchangeably.

3.4.3.3.1 Queue: Definition

A *queue* is a data structure for maintaining a sequential list of objects.

There are three types of managed queues:

- Packet Buffer queue. The items are packet buffers and are linked together using the first field in the `paktype` structure.
- “Data” queue. The linkage structure is external to the items themselves and is managed internally by the scheduler. This implementation has more overhead, both in CPU cycles and in memory, but provides two advantages:
 - The structures need not reserve the first field for linkage.
 - A structure may reside on multiple queues concurrently.
 - It provides some memory protection for the managed queue.
- Ordinary queue. If not the above, a queue of any type of structures where the items are linked together using the first field in the structure.

All types of managed queues have the following characteristics:

- The queue operates according to the first in first out (FIFO) rules of access.
- Major type of `QUEUE_EVENT`; minor type as specified at creation or later modified.
- Maximum number of elements. The maximum number of elements on the queue. When this limit is reached the item is not enqueued and an error is returned. If a zero is specified, the number of elements is not limited by element count.

3.4.3.3.2 Queue Operations

Event Management provides the following functions to manage queues:

1 `create_watched_queue()`

Creates a queue whose watchers can be notified when new queue elements are added.

2 `process_set_queue_minor()`

The specified value is stored into the minor event type field of the specified queue.

3 `process_watch_queue()`

Creates, destroys, or alters the calling task’s notification event for the specified queue.

4 `process_set_queue_first_only()`

Specifies, when notification is required, that only the first task in the watch list is to be scheduled. In the case of multiple watchers, the first is scheduled and its event is moved to the end of the watch list and the second event becomes the first. The operation causes round-robin scheduling of the watchers.

5 `process_enqueue()`

Adds the specified element to the end of the specified queue and generates an event that causes tasks watching the queue to be scheduled. Tasks are scheduled in the order that they appear on the watch list.

6 process_unqueue()

Removes an element from an “ordinary” queue. Element removal does not cause an event to be generated or any task to be scheduled.

7 process_requeue()

Adds the specified element to the beginning of the specified queue and generates an event that causes tasks watching the queue to be scheduled. Tasks are scheduled in the order that they appear on the watch list.

8 process_inqueue()

Adds the specified element to the middle of the specified queue and signals an event that causes tasks watching the queue to be scheduled. Tasks are scheduled in the order that they appear on the watch list.

9 process_enqueue_pak()

Adds a packet buffer to the end of the packet buffer queue and signals an event that causes the watchers of the packet buffer queue to be scheduled. Watchers are scheduled in the order that they appear on the watch list.

10 process_requeue_pak()

Adds a packet buffer to the beginning of the packet buffer queue and signals an event that causes the watchers of the packet buffer queue to be scheduled. Watchers are scheduled in the order that they appear on the watch list.

11 process_data_enqueue()

When an external data pointer is required, either because the structure does not have the first field reserved for a pointer or because the structure may be placed on multiple queues, use the `process_data_enqueue()` function. If the item cannot be enqueued, this function returns FALSE.

12 process_dequeue()

Removes the first item from the specified queue and returns its pointer. This function processes both data queues and packet buffer queues.

13 process_data_dequeue()

Removes the first item from the specified data queue and returns its pointer.

14 process_peek_queue()

Returns a pointer to the first item on the queue without removing it from the queue.

15 process_queue_size()

Returns the current number of elements in the specified queue.

16 process_queue_max_size()

Returns the maximum number of elements that can be stored in the specified queue.

17 process_resize_queue()

Replaces the maximum number of queue elements with the specified new maximum. If that value is less than the current number of queue elements, the queue is processed from the first element to the last in order. All elements whose position is greater than the new specified maximum are not deleted, but are placed on an overflow queue.

18 process_is_queue_full()

Returns TRUE if the specified queue is full; otherwise, returns FALSE.

19 process_is_queue_empty()

Returns TRUE if the specifies queue is empty; otherwise, returns FALSE.

20 delete_watched_queue()

Unlinks and frees the data structures for any tasks that are watching the queue and then frees the queue data structure itself. The function also clears the input pointer to prevent dangling pointers. Caution: This function does not free individual queue elements. Therefore, it should only be used on an empty queue.

3.4.3.4 Managed Booleans

This section describes the characteristics of managed Booleans.

3.4.3.4.1 Managed Boolean: Definition

A *Boolean* is a variable of integer type whose truth value is FALSE if it is zero and TRUE if it is nonzero. A *managed Boolean* is a Boolean whose transition from FALSE to TRUE causes its watchers to be scheduled.

Managed Booleans are supported with the following functions:

1 create_watched_boolean()

Creates a new managed Boolean with a given name and minor event type.

2 process_set_boolean_minor()

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 task to unambiguously re-identify Booleans created in this way before watching them.

3 process_watch_boolean()

Notifies the scheduler of a change in the caller's dependence on the watched Boolean specified by *wb*. The caller may either enable or disable dependence by setting the *enable* argument to ENABLE or DISABLE respectively. The caller may also specify whether the dependence is a one time dependence or recurring dependence by setting the *one_shot* argument to ONE_SHOT or RECURRING respectively.

The caller's event will be notified when the specified Boolean transitions from FALSE to TRUE.

4 process_set_boolean_first_only()

Specifies how many tasks are to be scheduled when a Boolean event is generated for the given managed Boolean. If *first_only* is set to FALSE, all watchers are scheduled. If *first_only* is set to TRUE, only the first watcher is scheduled. In the latter case the scheduled watcher is moved to the end of the watch list.

5 process_set_boolean()

The value of the watched Boolean specified by *wb* is replaced by the value specified by *value*. In addition the old and new values of *value* are compared in an attempt to determine if a Boolean event should be generated. Table 3-5 indicates the conditions under which an event is generated:

Table 3-5 Boolean Event Generation

Old	New	Event Generated
F	F	No
F	T	Yes
T	F	No
T	T	No

6 process_get_boolean()

Returns the value of the given managed Boolean.

7 delete_watched_boolean()

Deletes the watched Boolean specified by *wb*. It does so by first unlinking and freeing data structures for any tasks that are watching the specified queue and then frees the Boolean data structure itself.

3.4.3.5 Managed Bit Fields

This section describes the characteristics of managed bit fields.

3.4.3.5.1 Managed Bit Fields: Definition

A *bit field* is a 32-bit quantity of integer type in which each of the individual bits has significance.

Additionally, a *managed bit field* (also called a watched bit field) is a bit field that when changed causes one or more watchers to be scheduled.

Managed bit fields are supported by the following functions:

1 create_watched_bitfield()

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

2 process_set_bitfield_minor()

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 task to unambiguously re-identify bit fields created in this way before watching them.

3 process_watch_bitfield()

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

4 process_set_bitfield_first_only()

If several tasks are watching a single bit field and you want only one task to be awoken for each change, use the `process_set_bitfield_first_only()` function.

The scheduler keeps a list of all tasks watching a particular bit field. Upon change to the bit field, the first task in that list is scheduled to execute and the task is then placed at the end of the list. This results in a round-robin execution of tasks as enqueues occur.

This function is not frequently used as bit fields are normally only watched by a single task.

5 `process_get_wakeup()`

When awoken, a task determines that a managed bit field has been changed when the major type as returned by `process_get_wakeup()` is set to `BITFIELD_EVENT`.

6 `process_get_bitfield()`

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

7 `process_set_bitfield()`

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

Additionally, if the argument `send_wakeup` is set, all tasks watching this bit field are scheduled to run; if the argument `send_wakeup` is not set, the change is made and the watchdog tasks are not scheduled.

8 `process_clear_bitfield()`

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.

Additionally, if the argument `send_wakeup` is set, all tasks watching this bit field are scheduled to run; If the argument `send_wakeup` is not set, the change is made and the watchdog tasks are not scheduled.

9 `process_keep_bitfield()`

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.

Additionally, if the argument `send_wakeup` is set, all tasks watching this bit field are scheduled to run. If the argument `send_wakeup` is not set, the change is made and the watchdog tasks are not scheduled.

10 `delete_watched_bitfield()`

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 tasks that are watching this bit field, and then frees the bit field data structure itself. The function also clears the input pointer to prevent dangling pointers.

3.4.3.6 Managed Semaphores

This section describes the characteristics of managed semaphores.

3.4.3.6.1 Managed Semaphore: Definition

A *watched semaphore*, or *managed semaphore*, contains an atomic lock 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`.

A managed semaphore can be used both at process level and interrupt level. However, at interrupt level, no delay time is allowed. Semaphores cannot be unlocked from interrupts. At process level, you may supply a maximum time to delay waiting to obtain the lock to the resource set.

Note Use a managed semaphore when you require that a process block, waiting for the lock to be available when it is released.

Managed semaphores are supported by the following functions:

1 `watched_semaphore_get_name()`

To get the name of a watched semaphore, call `watched_semaphore_get_name()`.

2 `watched_semaphore_get_owner()`

To get the owner PID of a watched semaphore, call `watched_semaphore_get_owner()`.

3 `create_watched_semaphore()`

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

4 `create_watched_recursive_semaphore()`

This function extends the `create_watched_semaphore()` function to allow the current owner to acquire the semaphore multiple times. The `create_watched_recursive_semaphore()` function provides general support for recursive semaphores, which are semaphores that allow the owner to acquire the lock multiple times without deadlocking.

5 `create_watched_priority_semaphore()`

The `create_watched_priority_semaphore()` function creates and initializes a data structure with a property of prioritization that is managed by the system. Any process can register for notification when the state of this semaphore changes. The process is automatically scheduled for execution when the semaphore is released. The prioritization property permits that only one process will be woken up from the list of waiting processes for that semaphore. That process will be the highest priority process among the list of waiting processes. If more than one process with the same priority is found, a “Round Robin” scheme would be used to wake up the processes, with the longest-waiting process being permitted to run before other processes of the same priority.

6 `process_set_semaphore_minor()`

Cisco IOS software provides for minor types for semaphores as this field exists in all `watched_item` structures, therefore the `process_set_semaphore_minor()` function is available to modify the minor type that the semaphore receives when the semaphore state changes. However, as semaphores are usually locked directly, this is not often used.

7 `process_lock_semaphore()`

To guarantee the atomicity of operations that modify its resources, you must lock the semaphore before making the modification and unlock the semaphore when the modification is complete.

Use the `process_lock_semaphore()` to request the lock on the resource set, specifying a maximum timeout to wait to obtain the lock. A TRUE return indicates the lock is obtained; a FALSE return indicates the lock could not be obtained in the time specified. At interrupt level a delay time is not permitted as interrupt-level routines cannot wait.

8 `process_unlock_semaphore()`

Use the `process_unlock_semaphore()` function to unlock a semaphore.

9 `process_set_semaphore_first_only()`

The scheduler keeps a list of all tasks watching a particular semaphore. In normal operation, an unlock of the semaphore causes all tasks waiting on that semaphore to be scheduled to execute. The highest priority task runs first and grabs the lock; the others fail to get the lock and sleep again.

Using `process_set_semaphore_first_only()` alters the flow so that only the first task on the list is run, causing ordering of the semaphore to be in time-request order rather than task-priority order. To control the ordering, set the `first_only` Boolean appropriately:

10 `delete_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 tasks that are watching this semaphore, and then frees the semaphore data structure itself. This function also clears the input pointer to prevent dangling pointers.

3.4.3.7 Atomic Locks

An *atomic lock* is a single memory location that can be set or cleared by routines that function atomically; it does not use the event manager managed object notification and execution facilities. It is represented by the `semaphore` data structure. To guarantee the atomicity of operations that modify these resources, a function must lock the atomic lock before making the modification and unlock the semaphore when the modification is complete.

Since the structure for an *atomic lock* is smaller than for a *managed semaphore*, *atomic locks* may be selected when the number of structures is large.

Note Use atomic locks when you do not require the ability to block until the lock is released. Atomic locks simply return success or failure of the lock request.

Atomic locks are supported by the following functions:

1 `lock_semaphore()`

For an atomic lock, use the `lock_semaphore()` to attempt to obtain the lock. If the lock is not obtained, either wait (for instance, `process_suspend()`) or return an error.

2 `unlock_semaphore()`

Use `unlock_semaphore()` to unlock the semaphore.

3.4.3.7.1 Locking and Unlocking an Atomic Lock: Example

No code outside of the IOS event-handler code should be calling `lock_semaphore()` and `unlock_semaphore()` directly.

3.4.3.8 Read/Write Locks

Read/Write locks, as the name implies, allow multiple execution units to share the read operation of data, whereas only one execution unit will be allowed to modify the protected data at any given instant of time. (*New in 12.3T*)

Read/write locks are supported by the following functions:

1 `create_watched_rwlock()`

Use the `create_watched_rwlock()` function to create, initialize and return a pointer to a `watched_rwlock` abstract data type.

2 `delete_watched_rwlock()`

Use the `delete_watched_rwlock()` function to destroy the `watched_rwlock` abstract data type pointed to by `rwlock`.

3 `process_rwlock_rdlock()`

Use the `process_rwlock_rdlock()` function to obtain a shared read lock for the calling process.

4 `process_rwlock_rdlock_timed()`

Use the `process_rwlock_rdlock_timed()` function to acquire a lock before the specified timeout parameter.

5 `process_rwlock_unlock()`

Use the `process_rwlock_unlock()` function to release the lock obtained by previous calls to either `process_rwlock_rdlock()` or `process_rwlock_wrlock()`.

6 `process_rwlock_wrlock()`

Use the `process_rwlock_wrlock()` function to obtain an exclusive “write” lock for the calling process.

7 `process_rwlock_wrlock_timed()`

Use the `process_rwlock_wrlock_timed()` function to obtain an exclusive “write” lock for the calling process before the specified timeout parameter.

3.4.3.9 Managed Messages

A *message* is a simple communications mechanism passing only an ID, numeric, and a pointer. The advantage of messages over callbacks is that the code executes in the context of the receiving task, allowing for execution at the priority of the receiving task. A standard message is always delivered immediately; watched messages can be disabled by ID.

Note Regarding messages sent from interrupt level, both simple and managed messages use dynamically-expandable standard chunks to hold the messages to be delivered. Because standard chunks do not have “pre-depletion” expansion, sending a message at interrupt level may fail due to lack of chunk elements at that time.

Managed messages are supported by the following functions:

1 `process_send_message()`

To send a simple or a managed message to a task, use the `process_send_message()` function. The calling code segment need not be aware whether the receiving task is treating this message as a simple message or a watched message:

Note If you are passing structures between tasks in the same subsystem you can use a managed queue. Managed queues work off a common pointer defined in the subsystem. If your tasks (or interrupt code and a process) are in different subsystems and do not have a common address in which to store the pointer to the managed queue, you will have to use messages.

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

For example, to find the time spent between a message event posted and processed, you can take a timestamp, embed it in the data structure that you send via `process_send_message()` and take a delta with the timestamp using the `ELAPSED_TIME64()` or `ELAPSED_TIME()` macro when you dequeue it.

2 `process_get_wakeup()`

When awoken, a task determines that a simple message has been queued when the major type as returned by `process_get_wakeup()` is set to `MESSAGE_EVENT`.

When awoken, a task determines that a watched message has been queued when the major type as returned by `process_get_wakeup()` is set to `WATCHED_MESSAGE_EVENT`.

3 `process_get_message()`

To retrieve the next message that is waiting for this task, use the `process_get_message()` function, that fills in all three arguments.

4 `process_watch_watched_message()`

To register the task to be notified when a watched message is delivered, use the `process_watch_watched_message()` function. In this function, you specify the message ID and whether notification is being enabled or disabled. If you are performing an enable, also specify the routine to run for cleanup when a disable is issued.

When a `process_watch_watched_message()` is received with a disable state, the cleanup routine is run for each message of that `message_id` and then the memory associated with that message is returned.

When a `process_watch_watched_message()` is received with an enable state, a cleanup routine is specified. Then when a `process_watch_watched_message()` with disable state is received, a `process_pop_event_list()` or a `process_kill()` is issued, the cleanup routine is run for each message of that `message_id` and then the memory associated with that message is returned.

5 `process_get_watched_message()`

To retrieve the next watched message that is waiting for this task, use the `process_get_watched_message()` function.

3.4.3.10 Managed Timers

Many operating systems use a hardware decrementing timer to indicate timer expiration. With many tasks using timers, this has high overhead. Since the Cisco IOS software is a nonpreemptive system anyway, this additional overhead is superfluous. Instead the Cisco IOS scheduler uses the following

technique: as each task gives up control of the processor (either to suspend or go idle) the scheduler looks for tasks that have an expired timer. Each task with an expired timer is moved to the end of its appropriate ready list. This has implications for developers wanting real-time behavior.

The purpose of these functions is to suspend for a finite period of time, also referred to as “sleeping”.

The effect of the functions that are available for waiting on timers is to execute a task again as soon as the specified time has been reached or as soon as the time interval has elapsed. The task does not wake up for any other reason.

Managed timers are supported by the following functions:

1 process_sleep_for()

Sleep for the specified amount of time (number of milliseconds) from the current time.

2 process_sleep_on_timer()

Sleep until a managed timer expires. This is the preferred method if the task uses managed timers or if the timers within the task are modified by any other task. The wakeup time of the modified managed timer percolates from the task’s private timers up into the scheduler, thereby automatically adjusting the wakeup time of the task.

3 process_sleep_periodic()

Sleep for a specified amount of time (number of milliseconds) from the last time the task was awakened. This allows tasks to be scheduled at a fixed time interval, even though a single execution might have been delayed because the processor was being used for other work.

4 process_sleep_until()

Sleep until an absolute time is reached (based on boot having a time of 0).

5 process_safe_may_suspend()

Safely sleep for the specified amount of time (number of milliseconds) from the current time.

6 process_safe_sleep_on_timer()

Safely sleep until a managed timer expires. This is the preferred method if the task uses managed timers or if the timers within the task are modified by any other task. The wakeup time of the modified managed timer percolates from the task’s private timers up into the scheduler, thereby automatically adjusting the wakeup time of the task.

7 process_safe_sleep_periodic()

Safely sleep for a specified amount of time (number of milliseconds) from the last time the task was awakened. This allows tasks to be scheduled at a fixed time interval, even though a single execution might have been delayed because the processor was being used for other work.

8 process_safe_sleep_until()

Safely sleep until an absolute time is reached (based on boot having a time of 0).

9 process_watch_mgd_timer()

To register a task to be notified that a timer has expired, use the `process_watch_mgd_timer()` function for managed timers. (Timers are discussed in the “Timer Services” chapter.) To DISABLE timer expiration notification, set the `enable` argument to DISABLE.

When the timer expires, the task that is watching it is awakened. A task 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 task can watch only one managed timer directly, an arbitrary number of managed timers might be subordinate to that single watched managed timer.

The wakeup time of a managed timer can be changed by another code section having access to that timer while a task is sleeping, and the new wakeup time automatically propagates into the scheduler.

Multiple tasks cannot watch the same managed timer. Whenever a task waits while watching a managed timer, that timer is linked into a node that is unique to that task in the scheduler's own timer tree. Allowing two tasks to wait on the same managed timer would require linking a single managed timer into two different nodes of the scheduler's timer tree at the same time. This cannot be done.

10 process_watch_timer()

To register a task to be notified that a timer has expired, use the `process_watch_timer()` function for simple (passive) timers. (Timers are discussed in the “Timer Services” chapter.) To DISABLE timer expiration notification, set the `enable` argument to DISABLE.

When the timer expires, the task that is watching it is awakened.

For simple timers, the wakeup time is read when the task relinquishes the CPU and cannot be dynamically updated once the task is asleep. In contrast, the wakeup time of a managed timer can be changed by another code section having access to that timer while a task is sleeping, and the new wakeup time automatically propagates into the scheduler.

3.4.4 Event Management Topics

This section describes some Event Management topics.

3.4.4.1 Directly Waking up a Cisco IOS Task

In almost all cases, task-wake up occurs as an adjunct to or by-product of another task management event, such as a managed timer, a managed semaphore, or the like. However, there are cases where direct task wake up are valid. Most of these involve the waking of tasks that directly manage non-networking devices like flash memory or that change the operational status of a routing protocol. For these and other such cases, two functions, `process_wakeup()` and `process_wakeup_w_reason()` allow tasks to be awakened directly. The two differ only in the reason code that is communicated to the awakened task.

To wake up a task directly using the default reason of DIRECT_EVENT, use the `process_wakeup()` function.

```
void process_wakeup(pid_t pid);
```

To wake up a Cisco IOS task directly supplying a reason, use the `process_wakeup_w_reason()` function.

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

Note The `process_wakeup()` function and the DIRECT_EVENT wakeup reason are being deprecated in Releases 12.3 and above, so avoid using this reason and migrate the code to using a watched event instead. See Section 3.4.3, “Managing Individual Events”, for more information about individual types of watched events.

3.4.4.2 Determining Why a Task was Awakened

To determine the classes of events that can awaken the current task, use the `process_get_wakeup_reasons()` function.

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

To determine the next or only reason that the current task was awakened, use the `process_get_wakeup()` function.

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

3.4.4.3 Waiting on Events

To wait until events occur, use the following functions:

- 1 `process_wait_for_event()`
- 2 `process_wait_for_event_timed()`

The thread of the calling function is placed on the idle queue where it stays until it is explicitly awakened. As previously noted, such awakening may be either nonspecific via `process_wakeup()` (which is actually being deprecated in Release 12.3 and above) or it may be specific via `process_wakeup_w_reason()`.

The only managed object that is different is the semaphore object. Otherwise, the managed objects are all the same with regards to waiting for an event. The semaphore object always attempts to take a lock, so it immediately waits for the event.

These functions are described below.

Note The `process_wait_for_event()` function will go through the list of events that the process is waiting on, and if `process_wait_for_event()` discovers an event that has not been handled, `process_wait_for_event()` arranges for the process to be put onto the run queue with that event pending and for `wi_occurred` to be set to TRUE if necessary. This is done inside the `process_events_waiting_p()` function. The major caveats are:

1.
The scheduler stops this process with the first unhandled or partially-handled event that it finds, because that is all the scheduler needs in order to put a process onto the run queue. The scheduler makes no attempt to re-issue all unhandled events, so you need to make sure that your code does not suspend with unhandled or partially-handled events every time.
2.
The scheduler cannot tell by examining the bitfield whether or not a watched bitfield event has been handled, so you should never suspend with a partially-handled bitfield event.

You can call `process_suspend()`, `process_may_suspend()`, or `process_sleep_for()` with partially-handled bitfield events if you need to give up the CPU for a time while in the midst of handling the bitfield event. However, do not call `process_wait_for_event()` with a partially-handled bitfield event because your code might never wake up again.

If your code “suspends” while processing one type of event, that is just fine because this is what most of the code does.

If your code processes only a few events of a given type and then calls `process_get_wakeup()`, you will get the next type of event (if any).

If there are none and you try to “block”, the scheduler will check it for you.

The scheduler will figure out that your code did not finish the events and will keep your code in the ready queue itself instead of moving your code to idle queue.

In fact, this may be done to process different types of events “fairly”—to give some time to queue events then some time to timer events, and so on. For example:

```
while (1) {
    process_wait_for_event();
    while (process_get_wakeup(&major, &minor)) {
        switch (major) {
            case TIMER_EVENT:
                handle_timer_event();
                break;
            case QUEUE_EVENT:
                while (!QUEUEEMPTY(&queue)) {
                    event = process_dequeue(queue);
                    do_queue_event(event);
                    <suspend if necessary>
                }
                break;
            case BOOLEAN_EVENT:
        }
    }
}
```

If the above code always has queue elements, then it may never get a chance to process timer events.

For this reason, to be fair between different type of events, you can write code like this:

```
while (1) {
    process_wait_for_event();
    while (process_get_wakeup(&major, &minor)) {
        switch (major) {
            case TIMER_EVENT:
                handle_N_timer_events();
                break;
            case QUEUE_EVENT:
                process_N_elements_from_queue();
                break;
            case ...
        }
    }
}
```

The above code tries to process queue and timer events fairly, but the caveats mentioned in the note above will kick in. This means that there may be latency to certain events, but eventually everything will get processed.

3.4.4.3.1 process_wait_for_event()

Suspend the task until an asynchronous event occurs, also referred to as “waiting”.

When you call `process_wait_for_event()`, the scheduler goes through all of the events that the process is watching. If the scheduler finds, for example, a watched queue that still contains some elements, it arranges for the process to be put directly onto the run queue with a `QUEUE_EVENT` pending.

Some possible issues still remain:

- 1 The scheduler doesn't exhaust the list of events the process is watching; instead, the scheduler stops looking when it finds the first pending event for the process. So, for example, if you are watching multiple queues and never clear any of them out, the scheduler may keep giving you events for the same queue (the first one in your event list) and ignore the other queues.
- 2 The scheduler checks first for timer events, then for messages, then for all other events, so if you always have a timer event pending when calling `process_wait_for_event()`, it will not check the queues to see if there is a queue event pending.
- 3 The scheduler cannot tell by examining the state of a watched bitlist whether there are any unhandled bit changes there, so it cannot synthesize watched bitlist events for you when you call `process_wait_for_event()`. This means you must always completely handle watched bitlist events before calling `process_wait_for_event()`.

This amounts to a guarantee (with the exception of watched bitlists) that if for each event the process is watching, at some point it completely handles that event, then it will eventually be given enough events of the correct types to cause all the events it is watching to be processed. This is true even if occasionally it calls `process_wait_for_event()` without completely handling a watched event that it has received a notification for.

Or to put it another way, if `process_wait_for_event()` puts a process on the idle queue and not the run queue, then (with the exception of watched bitfields) the scheduler guarantees that the process has handled all of its events and it really does need to wait for another event to happen before it is run again.

Thrashing

Basically, every Cisco IOS process, when woken up, checks its queue of things to do and performs those tasks. Every process runs in a loop trying to do as much as it can, without exceeding a certain time limit. If a process exceeds its allocated time (about 2 seconds), the scheduler complains that it is a CPU HOG. The opposite of this is thrashing. When a process still has things to do, but gives up processing them too quickly, it gets rescheduled to run again. If that happens too often, it is called thrashing.

The following example is one such example of thrashing.

The SCHED-3-THRASHING message indicates that the process called `process_wait_for_event()` while there were still some events on the process queue. And that this happened for more than `SCHED THRASH_THRESHOLD` (which is 50 in 12.2T) times.

In the code below, you may be breaking out of the inner loop `while (process_get_wakeup(&major, &minor){})` prematurely (that is, without processing all the events present on the queue).

```
while (TRUE) {
    process_wait_for_event();
    while (process_get_wakeup(&major, &minor)) {
        ...
        if (...) {
            break;
        }
        process_may_suspend(); /* don't hog the CPU */
    }
    ...
}
```

3.4.4.3.2 process_wait_for_event_timed()

Suspend the task until an asynchronous event occurs, providing a temporary timer to limit how long the task can remain idle.

3.4.4.4 Waiting Until System Initialization Completes

Under most circumstances, IOS tasks are designed to operate in a system whose infrastructure has been initialized, that is, one in which all or most of the kernel and the low level infrastructure services have prepared themselves for use by callers. However, during system-wide initialization, such preparation has not yet occurred.

The functions that are provided to allow a task to synchronize its operation with system initialization events are `process_wait_on_system_init()` and `process_wait_on_system_config()`.

The two waits are sequential. The `system_configured` flag gets set first, then the `systeminit_flag`. The `systeminit_flag` means that system initialization is done.

The `process_wait_on_system_init()` function blocks a task until the global Boolean `systeminit_flag` is set to TRUE. When this happens, all the configuration processing is complete (including processing of network-resident configuration files) and the `(*idb->system_configured)` vector has been called for all hardware IDBs.

This function never returns if called in the context of a boot image (`system_loading ==TRUE`).

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

The `process_wait_on_system_config()` function blocks a task until the global Boolean `system_configured` is set to TRUE. This occurs after all NVRAM-based configuration file processing is complete, but before processing of any configured network-resident configuration files has started.

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

3.4.4.4.1 Wait Until System Initialization Completes: Example

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

```
void my_network_protocol_input (void) {
```

Any necessary memory allocation and initialization can occur at this point. However, a module may not want to rely on the fact that memory is initialized after other peer modules. That is why the `system init` flag exists to synchronize further initialization.

```
sys/ip/ip_nat.c:

static process
ipnat_ager (void)
{
    ulong major, minor;

/*
 * NAT configuration may send messages with work that needs
 * to be done after initialization is complete, so wait for this
 * before continuing.
 */
process_wait_on_system_init();
```

```

process_watch_mgd_timer(&ipnat_timer_master, ENABLE);
while (TRUE) {
    process_wait_for_event();
    while (process_get_wakeup(&major, &minor)) {
        switch (major) {
            case TIMER_EVENT:
                ipnat_ager_timers();
                break;
            case MESSAGE_EVENT:
                ipnat_ager_messages();
                break;
            default:
                errmsg(&msgsym(UNEXPECTEDEVENT, SCHED), major, minor);
                break;
        }
    }
}
[...]

```

3.4.4.5 Disabling Blocking and Using Safe Blocking with process_safe_*() Functions

Warning Improper disabling of blocking can result in a “CPU HOG” condition that eventually results in the task being terminated. Use the `process_safe_*`() functions described in this section with great care.

By default, all functions may call one of the task event management or suspension functions, and suspend the task in which they are executing. However, there are special cases when a task must disable the ability of the function it calls to issue a task suspend or task wait call. This happens most often for a time-critical task that needs to run to completion. Some examples where tasks might do this include OIR (Online Insertion & Removal), Packet Memory Setup, and Multi-Link PPP Bundle Reset. The following function disables task blocking by setting the `blocking_disabled` flag, and returns the previous blocking state (contents of that flag) to the caller:

```
boolean set_blocking_disabled (void)
```

If blocking was already disabled, this function returns TRUE.

To set the `blocking_disabled` flag to a new value, call the `reset_blocking_disabled()` function, passing the value as the parameter (usually called to restore the “old” value):

```
void reset_blocking_disabled (boolean old);
```

Once the `set_blocking_disabled()` function has been issued, all subsequent calls to `process_suspend()`, `process_wait_for_event()`, and other process blocking functions do not result in suspension. A task will continue executing until either the `blocking_disabled` flag is reset or one of the `process_safe_*`() functions is called.

Disabled blocking can be used in two ways. The most commonly discussed use is in a low level, short-running section of code that performs a critical function that must not relinquish control to another task. In practice, disabled blocking is more commonly used (but much less commonly discussed) in relatively complex thread-level sets of operations that, in general, need to operate without interruption except at specific points. At such points, control must be relinquished and the task must block, perhaps to wait for another task completion. The recommended implementation in this situation would be to disable blocking at the start of the relevant section of code, and enable it when a blocking call must be issued, as shown in the following code example:

```

old_blocking_state = set_blocking_disabled(); // This is the blanket block
    <. . . perform required protected actions >
    reset_blocking_disabled(old_blocking_state);
    process_wait_for_event(); // or whatever other blocking call is required
    old_blocking_state = set_blocking_disabled();
    <now blocking is disabled and the thread is again protected>

```

process_safe_* Functions

The `process_safe_*`() blocking functions allow the caller to invoke a blocking call when blocking is assumed to be disabled (but doesn't necessarily have to be so). With the code updates committed in CSCsj24186, these functions preserve whatever the current blocking state is whether or not blocking is disabled when they are called. The `process_safe_*`() functions provide the same functionality as their counterparts (same function name without "safe"), but save the current blocking status, allow process blocking for their process sleep-or-wait call, then restore the prior blocking state upon returning. The logic associated with `process_safe_*`() functions is as in the following example that uses the `process_wait_for_event()` call:

```

old_blocking_state = set_blocking_disabled(FALSE); // save cur state
process_wait_for_event(); // or other desired blocking call
reset_blocking_disabled(old_blocking_state); // restore old state

```

These functions allow a task to release the CPU even if the task blocking flag is disabled, but you should not rely on blocking to always be disabled upon returning from these calls (as might previously have been assumed). Developers should always explicitly call `set_blocking_disabled()` to disable task blocking/suspension. Here is a list of the "safe" blocking calls:

1 `process_safe_may_suspend()`

This function temporarily enables task blocking and releases the CPU so the task can be rescheduled via the `process_suspend()` Process Management call. After the task resumes execution, task blocking is restored to its previous state.

2 `process_safe_may_suspend()`

This function temporarily enables task blocking so that the task can sleep for a fixed period of milliseconds via the `process_sleep_for()` Event Management call. After the task awakens, task blocking is restored to its previous state.

3 `process_safe_sleep_on_timer()`

This function temporarily enables task blocking, adds the given managed timer into the list of the specified task's timer, and puts the task to sleep until one of the managed timers on the list expires. This is done via the `process_sleep_on_timer()` Event Management call. After the task awakens, task blocking is restored to its previous state.

4 `process_safe_sleep_periodic()`

This function temporarily enables task blocking and puts a task to sleep for a fixed period of milliseconds adjusted from its last wakeup time. This is done via the `process_sleep_periodic()` Event Management call. After the task resumes, task blocking is restored to its previous state.

5 `process_safe_sleep_until()`

This function temporarily enables task blocking and puts a task to sleep until a fixed period in time is reached. This is done via the `process_sleep_until()` Event Management call. After the task resumes, task blocking is restored to its previous state.

6 `process_safe_suspend()`

This function temporarily enables task blocking and releases the CPU so the task can be re-scheduled via the `process_suspend()` Process Management call. After the task resumes execution, task blocking is restored to its previous state.

7 `process_safe_wait_for_event()`

This function temporarily enables task blocking and suspends a task until an asynchronous event occurs. This is done via the `process_wait_for_event()` Event Management call. After the task awakens, task blocking is restored to its previous state.

Caution The “safe” blocking calls should only be used in thread-level code, not in any service-level code. In general, this would preclude their use in most of the IOS infrastructure.

3.4.5 Useful Event Management Commands

The following Event Management commands are important to know. For example, to find where you should check to see why a process doesn’t wake up after it has been sent messages, you can enable “service internal” in config mode and use the `show process event pid` command to see the events that the process is watching, and use the `show stack pid` command to see where the process is blocking.

show process event pid - This command displays all the events, as well as the status of the events, that this process is associated with. This command also says whether a semaphore is being watched by that process, its value, and the pid of the process holding the semaphore. For a semaphore event, owner information is printed too. If we see a process watching the semaphore alone and its *invoked* count from `show proc` never increases, we may say that it’s hanging on a semaphore.

show process event pid displays the pushed events information along with current events. In the following output, the watched events before “---” are the current events, and the ones after refer to the pushed events.

Router#**test scheduler library**

Issuing the command "show process events 103".

```
Process 103: Library Call Test, state: Waiting for Event
    Watching Boolean 'B flag' (0x61BCD824), id 0x80000002, value False.
    Watching bitfield 'test bitfield' (0x61BDE5F8), id 0x0, value 00000000.
    Watching queue 'Dummy queue' (0x61BBC894), id 0x0, count 0.
    Watching a delta timer. Expires in 00:02:00.
    ---
    Watching Boolean 'A flag' (0x61BCD880), id 0x80000001, value False.
    Watching Boolean 'Quit flag' (0x61BCD7C8), id 0x80000003, value False.
```

Setting Boolean A. Should not cause a wakeup.
Issuing the command "show process events 103".

```
Process 103: Library Call Test, state: Waiting for Event
    Watching Boolean 'B flag' (0x61BCD824), id 0x80000002, value False.
    Watching bitfield 'test bitfield' (0x61BDE5F8), id 0x0, value 00000000.
    Watching queue 'Dummy queue' (0x61BBC894), id 0x0, count 0.
    Watching a delta timer. Expires in 00:02:00.
    ---
    Watching Boolean 'A flag' (0x61BCD880), id 0x80000001, value True.
    Watching Boolean 'Quit flag' (0x61BCD7C8), id 0x80000003, value False.
```

Setting Boolean B. Should cause a wakeup.
Process level should have already handled Boolean A, or it

```
should be ready to run due to Boolean A being TRUE.  
Library test process awake: caller has events.  
Issuing the command "show process events 103".
```

```
Process 103: Library Call Test, state: Ready to Run
```

```
Process should exit.  
The process has terminated successfully.  
Router#
```

You can also use **show mem semaphore address** to find the pid.

show stack pid would be helpful to see what the process was doing, and hence whether it is blocked on a semaphore or not.

If you know the semaphore (or event) but not the corresponding process, then you can also use **show processes all-events** to see the watchers of this event. You can use **show processes all-events** to get the address of the event and do a memory dump of the same using the **show memory** CLI, which will also give the owner pid.

3.5 Random Number Generation

Random number generators that use deterministic algorithms are usually called pseudo-random number generators or pRNG. In spite of their randomness, some random numbers that are generated via a pRNG are “predictable.” Von Neumann said:

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

What did he mean? If you start with a seed (a number used to change the start of a sequence) and an algorithm, the entire sequence of numbers is predefined.

Within the Cisco IOS code base are two functions that produce random numbers that are “predictable,” which are available in all images:

- `random_gen()`
- `rand()`

For more information, see section 3.5.2, “Non-Cryptographic pRNG Functions.”

However, an application that is *cryptographically secure* requires that:

Given the nth number (or a certain number of previous numbers) in a sequence, the next number cannot be guessed.

This is accomplished by continually feeding in entropy (randomness) into the algorithm.

Cisco IOS provides two functions that produce random numbers that are cryptographically secure, which are available in all images:

- `random_fill()`
- `get_random_bytes()`

For more information, see section 3.5.3, “IOS Universal Cryptographic pRNG Functions.”

Cisco IOS provides two crypto functions that are only available in security (K9) images. For more information, see section 3.5.4, “Crypto Image-Only IOS pRNG Functions.”

This section contains the following subsections:

- Different Random Number Requirements
- Non-Cryptographic pRNG Functions
- IOS Universal Cryptographic pRNG Functions
- Crypto Image-Only IOS pRNG Functions
- Establishing an RNG Seed Context
- Establishing an RNG Seed Context: Examples
- Obtaining a Random Number
- Obtaining a Random Number: Examples
- Filling Memory with Random Numbers
- Filling Memory with Random Numbers: Examples
- Secure Hash Algorithm with `get_random_bytes()` Function
- Secure Hash Algorithm with `get_random_bytes()` Function: Example
- Further Information

3.5.1 Different Random Number Requirements

There are two fundamental uses of random numbers:

- Non-cryptographic—Very often, random numbers are needed for uses such as a small random time to delay, to provide a skewing, or test runs that are repeatable. For these common usages you need a random number, but it can be “predictable.”
- Cryptographic—for requirements such as session keys, initial sequence numbers, initialization vectors, salt values, and others, an unpredictable number is required.

This means using an algorithm that continually feeds in entropy, so that the sequence cannot be determined, that is, the $n+1$ th number cannot be determined from the previous.

Generating entropy is not easy, so use the functions provided by your operating system.

3.5.2 Non-Cryptographic pRNG Functions

Most of the applications just need a “random” number. It is not important that the next random number can be predicted.

Examples of usage are for jitters and delay times and non-crypto initial sequence numbers.

Two functions are predictable pRNGs within Cisco IOS:

- `random_gen()`—This function is IOS-specific, with a number of companion functions to provide customization. The full suite is:
 - `random_init()` and `random_init_context()`
 - `random_gen()` and `random_gen_context()`
 - `random_gen_32bit()` and `random_gen_32bit_context()`

The `random_gen()` functions are more efficient than the `rand()` function, so they may be called at both process and within interrupt level.

The suite of `random_gen()` functions are documented within the Cisco IOS API Reference at:

<http://wwwin-engd.cisco.com/ios/doc/pg/current/rsched.html>

- `rand()`—The `rand()` function is an ANSI-C standard function that returns the next non-cryptographically secure random number. Although there is no `rand()` function implementation within the IOS kernel, a number of applications implement the `rand()` function, both ANSI-C standard and non-standard implementations.

The `rand()` function is not terribly efficient, so it should only be called at process level. For more information, see the `rand()` reference page.

3.5.2.1 Cisco IOS RNG Facility

The Cisco IOS RNG Random Number Generation Facility provides the following features:

- A system-wide default as well as specific seed contexts with customizable seed parameters.
- A random return of values from 0 - 10,000 or a full 32-bits.
- Capability to fill in any number of bytes as requested.

3.5.3 IOS Universal Cryptographic pRNG Functions

In addition to the “predictable” pRNG available on all images, a set of non-predictable pRNGs is available on all IOS images.

They are used for cryptographically secure applications that are not restricted to the specific cryptographic (IP Sec) environment. Examples are challenge messages for AAA, RSVP, and SSH protocols, and salt values within cryptographic protocols.

These functions are not terribly efficient, so they should only be called from process level.

The two functions, which from your viewpoint as a coder may be considered equivalent, are:

- `random_fill()` function
- `get_random_bytes()` function

For detailed information about these functions, see their reference pages in the IOS API Reference at <http://wwwin-engd.cisco.com/ios/doc/pg/current/rsched.html>

3.5.4 Crypto Image-Only IOS pRNG Functions

Within crypto images only, there are two more pRNG functions. These are fairly slow, so they should only be used at process level:

- `rc4_fill()`—Used for IV and Unity message IDs
- `crypto_rng()`—Used for DSS keys

These functions are part of the crypto application and not part of the Cisco IOS Kernel and are not documented in the Cisco IOS API Reference. Before using, please contact the IOS crypto group at the `ios-crypto@cisco.com` email alias.

3.5.5 Establishing an RNG Seed Context

Note This information is IOS-specific to the `random_gen()` suite of functions and the `random_fill()` function. That is, the context is only for these functions. The `get_random_bytes()` function is independent and does not use this context information.

Two functions are available to establish the seed for a particular RNG, one for the system-wide default context, another for an independent RNG context. In either case, each invocation of this function will overlay the previous seed values.

To seed the system-wide default RNG, call the function:

```
void random_init(long s1_initial, long s1_initial);
```

To seed any other RNG context use:

```
void random_init_context(random_context *context,
                        long s1_initial, long s1_initial);
```

3.5.5.1 Establishing an RNG Seed Context: Examples

Upon system initialization, the main system startup routine will initialize the default context. It uses two pseudo-random numbers as seeds: the relative time since system boot in milliseconds and the checksum of the text area:

```
GET_TIMESTAMP32(curtime);
random_init((long)curtime, (long)textsum);
```

This may, for example, be later overridden if Kerberos Authentication is present on the system. This supplies more complex seeds, using the time since Jan 1 1970 (`uint seed`), a newly-established time since system boot and the same checksum:

```
cisco_kerb_srand(uint seed)
{
    GET_TIMESTAMP32(time);
    random_init(seed ^ time, textsum ^ seed);
```

CEF, Cisco Express Forwarding, has an independent context, `unique_id_rc`, which is used as the basis for all random number generation for this application, as exemplified below:

```
random_init_context(&unique_id_rc, fib_unique_id, ~fib_unique_id);
```

3.5.6 Obtaining a Random Number

Note This information is IOS-specific to the `random_gen()` suite of functions and the `random_fill()` function. That is, the context is only for these functions. The `get_random_bytes()` function is independent and does not use this context information.

After the RNG context has been established, random numbers can be obtained with a few variations:

To obtain a random number from 0 to 10,000 using the default context, call the function:

```
long random_gen(void);
```

To expand the range to a random 32-bit number, still using the default context, call the function:

```
long random_gen_32bit(void);
```

To obtain a random number from 0 to 10,000 using an independent context, call the function:

```
long random_gen_context(random_context *context);
```

To expand the range to a random 32-bit number, again using an independent context, call the function:

```
long random_gen_32bit_context(random_context *context);
```

3.5.6.1 Obtaining a Random Number: Examples

After any RNG context, either default or independent, has been established, a random number can be obtained.

The SPD Selective Packet Drop application uses a random number to determine whether a packet should be dropped:

```
static inline boolean ip_spd_random_drop (void)
{
    long random;
    random = random_gen();
    return ((random & 0xFF) < ip_spd.q_random_drop_probability);
```

The IOS timer function `timer_start_jitter()` uses a random 32-bit number in calculating the delay time:

```
ulong timer_calculate_jitter_common ( . . . )
{
. . . snip . . .
    jitter_seed = random_gen_32bit();
    jitter_value = jitter_seed % jitter_range;
    if (jittertype == MINUS)
        return (return (return (delay - jitter_value));
. . . snip . . .
```

TCP uses an independent context for a random number from 0 to 10,000, used later to generate a unique non-privileged port:

```
tcp_portseed = random_gen_context(&seed);
. . . snip . . .
firstport = lastport = port =(tcp_portseed << 9) | ttyseed;
```

CEF, Cisco Express Forwarding, uses the independent context established in the previous section and obtains a random 32-bit number to be used in the hash table for the entire CEF data base, the FIB Forwarding Information Base:

```
void ipfib_hash_algorithm_changed( . . . )
{
. . . snip . . .
    i1 = random_gen_32bit_context(&unique_id_rc);
    i2 = random_gen_32bit_context(&unique_id_rc);
    /*
     * Swap the two entries
```

```
*/
table_value = temp_hash_table[i1];
temp_hash_table[i1] = temp_hash_table[i2];
temp_hash_table[i2] = table_value;
```

3.5.7 Filling Memory with Random Numbers

Note This information is IOS-specific to the `random_gen()` suite of functions and the `random_fill()` function. That is, the context is only for these functions. The `get_random_bytes()` function is independent and does not use this context information.

To fill a supplied location and length with random numbers using MD5 Message Digest Algorithm, call the function:

```
void random_fill(uchar *dest, int len, boolean blockflag);
```

3.5.7.1 Filling Memory with Random Numbers: Examples

The RADIUS Remote Authentication Dial In User Service uses the random number fill function to create a random 16-byte RADIUS authentication string:

```
void radius_chap_login_internal ( . . . )
{
    random_fill(rad->rad_authent, RADIUS_AUTHENTICATOR_LEN, TRUE);
```

The CMTS Cable Modem Termination System uses the random number fill function to create a random 100-byte packet to be used in testing a link:

```
void cmts_send_random_frame (hwidbtype *hwidb, int count, boolean flag)
{
    random_pak = cmts_getbuffer(CR_ERR_RANDOMSIZE);
    if (! random_pak) {
        /* Handler "No packets available" error */
    }
    /*
     * Fill start of packet with random data
     */
    random_fill(random_pak->datagramstart, CR_ERR_BADTYPESIZE, FALSE);
```

3.5.8 Secure Hash Algorithm with `get_random_bytes()` Function

The `get_random_bytes()` function does not use the RNG context and is therefore independent. This function uses the SHA1 Secure Hash Algorithm to generate sets of 20 bytes of random numbers which are copied into the `*buf` buffer area:

```
void get_random_bytes(void *buf, uint nbytes);
```

This random field is generated from random data such as the clock delta since the last call and the PC of the last `malloc()` call, upon which the SHA1 hash is applied.

3.5.8.1 Secure Hash Algorithm with `get_random_bytes()` Function: Example

TCP uses the `get_random_bytes()` function to create a random 4 byte number to be used for an initial sequence number for a new TCP session:

```
ulong tcp_selectiss (void)
{
    ulong tmp;
    /*
     * RFC-793 wants the Initial Sequence Number to be tied to a
     * 32-bit clock whose low-order bit increments every 4ms or so.
     * Using clock based ISN is vulnerable to security attacks. The latest
     * RFC's on security RFC1948 & RFC1750 have different interesting
     * approaches to generate sequence numbers that are very hard to be
     * predicted by the intruders. The algorithm used here in
     * get_random_bytes
     * is partly based on a strong random number generator and the current
     * clock time and also an entropy factor using which the sequence numbers
     * generated are very hard to predict by the intruders.
     */
    get_random_bytes( &tmp, sizeof(long) );
    return ( tmp );
}
```

3.5.9 Further Information

Here are a few resources that discuss pRNGs. Within Cisco be sure to contact ios-crypto@cisco.com when making decisions about choosing the best function to suit your needs.

http://www.usenix.org/publications/library/proceedings/sec98/full_papers/gutmann/gutmann.pdf

<http://www.counterpane.com/yarrow-notes.html> (postscript only)

http://www.usenix.org/events/bsdcon02/full_papers/murray/murray.pdf

Memory Management

Added steps to show an example of how to debug a packet element leak to section 4.7.6.3.3 “show memory debug leaks chunks”. (October 2011)

Added websites for more information on memory management at beginning of chapter. (October 2011)

Added URL to section 4.7.2.3 “show memory dead Output”. (August 2011)

Added the CHUNK_FLAGS_NOHEADER_FAST flag to Table 4-8 “Chunk Pool Flags”. (December 2010)

Removed references to MEMPOOL_CLASS_FAST, MEMPOOL_CLASS_ISTACK, MEMPOOL_CLASS_PCIMEM, MEMPOOL_CLASS_PSTACK, and MEMPOOL_CLASS_MULTIBUS. (October 2010)

Added information about Garbage Detector in the Memory Management Command section 4.7.6 “Garbage Detector” (July 2010)

Added a note on the memory pointer setting by the application to section 4.3.10.1 “Example: Lock Memory”. (June 2010)

Corrected the space issue in the return commands (June 2010).

Added CHUNK_FLAGS_NONLAZY flag (New in 12.4(20)T) to Table 4-8 “Chunk Pool Flags”. Added note to section 4.4.3.1 “Example: Create a Memory Chunk” (July 2009)

Updated CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF flag (New in 12.2SR(autobahn76)) in Table 4-8 “Chunk Pool Flags”. (May 2009)

This chapter discusses regions, memory pools, and the chunk manager, and it describes the memory management functions in the Cisco IOS code. It also describes Virtual Memory (New in Release 12.0), Managed Chunks (New in Release 12.2) and Shared Information Utility (New in release 12.2SX).

Note Memory management development questions can be directed to the india-mem-team@cisco.com, mem-busters@cisco.com, and interest-mem@cisco.com aliases.

The following are websites for more information on memory management:

- IOS Memory Management Home Page—
<http://zed.cisco.com/confluence/display/OSII/IOS+Memory+Management+Home+Page>
- Memory Utilities—
http://zed.cisco.com/confluence/download/attachments/63367/memory_tools.ppt

Overview: Memory Management

- IOSMemory-DebugTools—http://wwwin-eng.cisco.com/cgi-bin/edcs/edcs_info?EDCS-990821
- NerdLunch - A life - Jacket for IOS(d) Memory Issues—<https://cisco.webex.com/ciscosales/lxr.php?AT=pb&SP=EC&rID=50069227&rKey=a5f4bd86872f645e>

The information in this chapter is divided into the following sections:

- Overview: Memory Management
- Regions
- Memory Pools
- Chunk Manager
- Managed Chunks (New in Release 12.2)
- Transient Memory Allocation (New in 12.2S)
- Memory Management Commands
- Dynamic Bitfield Management
- Dynamic Bitlists
- Virtual Memory
- Caching Issues
- ID Manager
- Shared Information Utility

4.1 Overview: Memory Management

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

- Regions and the Region Manager
- Memory Pools, Memory Pool Manager, and Free Lists
- Chunk Manager
- Transient Memory Allocation (New in 12.2S)

4.1.1 List of Memory Management Functions

The following functions are used for memory management (see the reference pages for detailed information on these functions):

- 1 `chunk_create()`
- 2 `chunk_destroy()`
- 3 `chunk_free()`
- 4 `chunk_is_destroyable()`
- 5 `chunk_lock()`
- 6 `chunk_lockable()`
- 7 `chunk_malloc()`

```
8 chunk_refcount()
9 chunk_totalfree_count()
10 free()
11 malloc()
12 malloc_aligned()
13 malloc_fast()
14 malloc_iomem()
15 malloc_iomem_aligned()
16 malloc_named()
17 malloc_named_aligned()
18 malloc_named_fast()
19 malloc_named_iomem()
20 malloc_named_iomem_aligned()
21 malloc_named_pcimem()
22 malloc_named_pcimem_aligned()
23 malloc_pcimem()
24 managed_chunk_create()
25 managed_chunk_destroy()
26 managed_chunk_free()
27 managed_chunk_malloc()
28 mem_lock()
29 memory_new_owner()
30 memory_specify_allocator()
31 mempool_add_alias_pool()
32 mempool_add_alternate_pool()
33 mempool_add_free_list()
34 mempool_add_region()
35 mempool_create()
36 mempool_find_by_addr()
37 mempool_find_by_class()
38 mempool_get_free_bytes()
39 mempool_get_total_bytes()
40 mempool_get_used_bytes()
41 mempool_is_empty()
42 mempool_is_low()
43 mempool_malloc()
44 mempool_malloc_aligned()
45 mempool_set_fragment_threshold()
46 mempool_set_low_threshold()
47 mem_refcount()
```

```
48 realloc()
49 realloc_named()
50 region_add_alias()
51 region_add_subalias()
52 region_create()
53 region_exists()
54 region_find_by_addr()
55 region_find_by_attributes()
56 region_find_by_class()
57 region_find_next_by_attributes()
58 region_find_next_by_class()
59 region_get_class()
60 region_get_media()
61 region_get_size_by_attributes()
62 region_get_size_by_class()
63 region_get_status()
64 region_init()
65 region_set_class()
66 region_set_media()
67 setstring()
68 setstring_named()
69 tm_malloc()
70 tm_malloc_aligned()
71 validlink()
72 validmem()
```

4.1.2 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.

Note The Transient Memory Allocation feature provides a new dynamic region manager to maintain a freeregionlist and provide memory (as regions) to memory pools as and when needed. A new transient mempool has also been added to serve the memory allocation requests that are transient in behavior. For more information about this feature, which is available on a limited number of platforms only, see section 4.6 “Transient Memory Allocation (New in 12.2S).”

4.1.3 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.

4.1.4 Chunk Manager

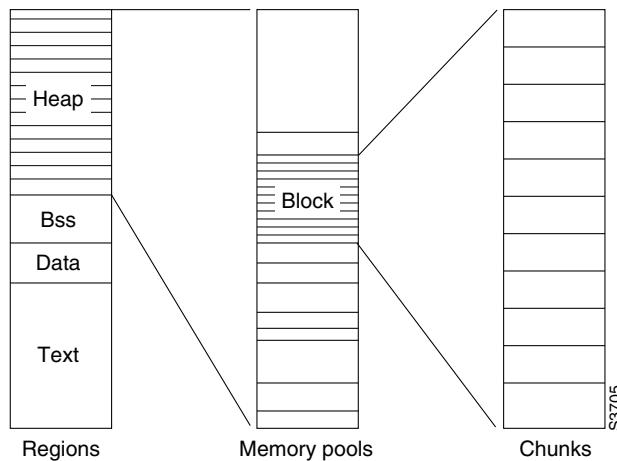
Having the memory pool manager provide comprehensive support for managing areas of memory requires an overhead of context for every block managed. When the memory pool manager is managing many thousands of small blocks, the overhead quickly becomes substantial. To avoid this overhead, some sections of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

Managed chunks were added in Release 12.2. From an external viewpoint, they allow a chunk to be expanded (acquire a new sibling) before the elements are all depleted. This is useful for chunks where elements are acquired at the interrupt level and success in allocation is important.

4.1.5 Relationship between Regions, Memory Pools, and Chunks

Figure 4-1 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.

Note One important aspect of Cisco IOS `malloc()` functions that is a deviation from ANSI standards is that the Cisco IOS `malloc()` functions all return memory that has been zero filled.

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.

Figure 4-2 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

REGION_CLASS_LOCAL "heap"
REGION_CLASS_IMAGEBSS "BSS"
REGION_CLASS_IMAGEDATA "data"
REGION_CLASS_IMAGETEXT "text"

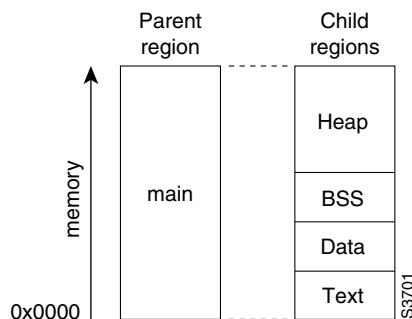
S3700

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.

Figure 4-3 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.

```
regiontype *region_create(regiontype *region, char *name, void *start,
                          uint size, region_class class, uint flags);
```

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 (see the `region_create()` reference page for the list of all classes). 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 Common 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) Memory accessible from the network interface PCI Bus, allowing for the direct transfer of packets from the network interface across the PCI Bus into “IO Memory”. Note that this memory type designation is only used for the c7200/7500 platforms and the platforms based on them. Other platforms that use PCI-accessible memory use the REGION_CLASS_IOMEM designation.

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 Hierarchy Types

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

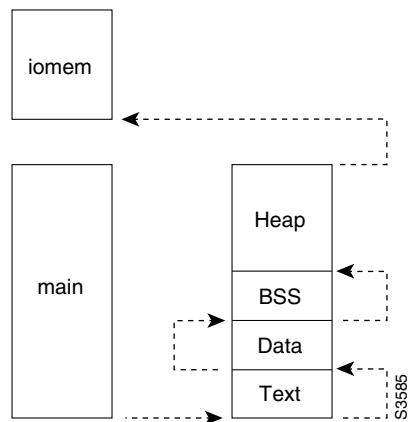
Table 4-3 Region Hierarchy Types

Type	Description
REGION_STATUS_PARENT	Parent region. These memory regions are completely unenclosed by 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

Figure 4-4 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 Establish a Subalias Region

To declare that a region of memory is an alias of a **portion** of an existing region, call the `region_add_subalias()` function.

```
#include "../os/region.h"
boolean region_add_subalias(regiontype *region, regiontype *alias,
void *alias_start);
```

4.2.10 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 region'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.

4.2.10.1 List of Region Inheritance Flags

Table 4-4 lists some possible region inheritance flags (see the `region_create()` reference page for the list of all 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's class.
REGION_FLAGS_INHERIT_MEDIA	Always inherit the parent's media type.
REGION_FLAGS_INHERIT_CLASS	Always inherit the parent's region class.

4.2.11 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_status status, region_media media);

regiontype *region_find_next_by_attributes(regiontype *region,
region_class class, region_status status, region_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.

4.2.11.1 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));
}
```

4.2.11.2 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);
}
```

4.2.12 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);
```

4.2.13 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_status status,
                                   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.

4.2.13.1 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);
```

4.2.14 Retrieve a Region's Attributes

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

Note Use the wrapper functions 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	<code>region_set_class()</code>	<code>region_get_class()</code>
Media	<code>region_set_media()</code>	<code>region_get_media()</code>
Status	—	<code>region_get_status()</code>

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 UNIX 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.

Note The Transient Memory Allocation feature provides a new dynamic region manager to maintain a freeregionlist and provide memory (as regions) to memory pools as and when needed. A new transient memory pool has also been added to serve the memory allocation requests that are transient in behavior. For more information about this feature, which is available on a limited number of platforms only, see section 4.6 “Transient Memory Allocation (New in 12.2S).”

4.3.2 Free Lists: Overview

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.

Note The Transient Memory Allocation feature provides a new transient memory pool to serve the memory allocation requests that are transient in behavior. There are also two new API functions for memory allocation from the transient memory pool: `tm_malloc()` and `tm_malloc_aligned()`. For more information about this feature, which is available on a limited number of platforms only, see section 4.6 “Transient Memory Allocation (New in 12.2S).”

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, const 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 `heapsizze` 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 heapsizze bytes,
 * and create a "local" memory pool based on it.
 */
region_create(&pmem_region, "heap", heapstart, heapsizze, 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 Common Memory Pool Classes

Table 4-6 lists the most common memory pool classes used by the system code (see the [mempool_create\(\)](#) reference page for a list of all classes).

Table 4-6 List of Common Memory Pool Classes

Class	Description
MEMPOOL_CLASS_LOCAL	(Mandatory) Main system heap.
MEMPOOL_CLASS_IOMEM	(Mandatory/aliasable) Shared memory for buffer data and controller descriptor blocks.
MEMPOOL_CLASS_TRANSIENT	(Optional) Memory for transient allocation. For more information, see 4.6.2 “Static and Transient Memory Pools.”

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.

```
extern void 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);
```

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.

```
extern void 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 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 Memory

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.

Note To expedite `malloc()` and to avoid memory fragmentation for memory blocks of 128 bytes or less on certain platforms like the 7200 there is a feature called "malloc lite" which is turned on by default. This feature gives a pre-allocated chunk when malloc'd for a block of 128 bytes or less. You cannot see this chunk if you use `malloc_named()` unless you turn off this "malloc lite" feature. You can turn this feature off by configuring **no mem try-malloc-lite**.

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(size_t 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(int mem_class, uint size, const char *name);
void *malloc_pcimem(int mem_class, uint size);
void *mempool_malloc(mempool_class class, uint size);
void *tm_malloc (size_t 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_malloc_aligned()` 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, uint alignment, const char
                                *name);
void *malloc_named_pcimem_aligned(int mem_class, uint size, uint align, const
                                  char *name);
void *mempool_malloc_aligned(mempool_class class, uint size, uint alignment);
void *tm_malloc_aligned (uint size, uint align)
```

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.	<code>malloc()</code>	<code>malloc_aligned()</code>
Allocates memory.	<code>malloc_fast()</code>	—
Allocates memory from MEMPOOL_CLASS_IOMEM.	<code>malloc_iomem()</code>	<code>malloc_iomem_aligned()</code>
Allocates memory from MEMPOOL_CLASS_LOCAL and binds a textual name to the allocated memory.	<code>malloc_named()</code>	<code>malloc_named_aligned()</code>
Allocates memory and binds a textual name to the allocated memory.	<code>malloc_named_fast()</code>	—
Allocates memory from MEMPOOL_CLASS_IOMEM and binds a textual name to the allocated memory.	<code>malloc_named_iomem()</code>	<code>malloc_named_iomem_aligned()</code>
Allocates memory and binds a textual name to the allocated memory.	<code>malloc_named_pcimem()</code>	<code>malloc_named_pcimem_aligned()</code>
Allocates memory.	<code>malloc_pcimem()</code>	—

Table 4-7 malloc() Family of Functions (continued)

Description	Function to Allocate General Memory	Function to Allocate Aligned Memory
Allocates memory from MEMPOOL_CLASS_TRANSIENT	tm_malloc()	tm_malloc_aligned()
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_malloc_aligned()

4.3.8.4 Guidelines for Allocating Memory

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

- 1 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.

Callers of `malloc()` must check for `NULL`, and take appropriate action. So what's appropriate action? It depends on the answer to "What are the consequences of not obtaining this memory (or storing a `NULL` pointer)?" Questions that may help decide include:

- (a) Does the customer lose something that will be retried?
- (b) Does the customer lose a feature, with a warning?
- (c) Could this lack cause incorrect routing behavior, possibly causing instability locally or in the entire network subdomain (routing area, autonomous system, etc.)?
- (d) Is this likely to happen again right away after a reboot?
- (e) Is this CLI processing code, where we could just say "can't do it"?

Another question might be "what does the customer want"? To handle this, we might have a function to warn of possible instability. It could crashdump if configured to do so by the user. (It is never recommended adding this in newly designed code, but it might be necessary to use when handling problems in existing code. New code should not be designed to crash! People who don't check `malloc` return codes are *designing their code to crash*.)

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.

- 2 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.

- 3 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;
}
```

This resulted in a huge amount of text segment space being used by these `errmsg()` (or sometimes `buginf()`) calls. So, 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
```

Note This is a rate-limited error message.

- 4 Memory allocation failures are reported by the memory management code via the `errmsg()` facility, so callers of `malloc()` should not report allocation failures because this would be redundant with the messages emitted by `malloc()`. For example, the following is neither necessary nor desired:

```
foo_context = malloc_named(sizeof(foo_context_t), "foo_context");

if (!foo_context) {
    errmsg(&msgsym(NOMEMORY, FOO), "foo structure");
    return;
}
```

Callers *should* report ancillary consequences of the failure if these consequences are something which would be visible and noteworthy to the user or administrator. The operator should not be left to wonder why some feature or command is not working, with the only hints being reports in the log about resource failures!

When reporting these ancillary failures, the programmer should use the appropriate method for printing that message according to the context where that failure occurs, following normal IOS guidelines for generating output. For example, if the failure occurs in the context of processing a command, then `printf()` normally should be used to report the ancillary failure, since the user at the console is (normally) the target of that message. If the failure occurs in other contexts, `errmsg()` or `buginf()` is more appropriate.

Although these ancillary messages should not directly report the `malloc()` failure, they should provide some hint that the secondary failure is a consequence of inability to obtain some required resource. This is particularly true when the output is directed to the operator at a console (that is, when using `printf()`), because that operator may never see the `errmsg()` emitted by the memory allocation system—bear in mind that the terminal may not be configured to receive log events. Even if the ancillary message is sent to the log, it is still useful to incorporate a hint into that message to help associate the event with prior events in the log. For example:

```
/*
```

```

        * An EXEC or CONFIG command ... talks to the user at a console.
        */
void foo_command (parseinfo *csb)
{
    foo_block = malloc_named(sizeof(foo_block_t), "foo_block");

    if (!foo_block) {
        printf("\nCan't do foo; insufficient system resources.");
        return;
    }
/*
 * An internal background function
 */
void foo_start (hwidbtype *hwidb)
{
    foo_context = malloc_named(sizeof(foo_context_t), "foo_context");
    if (!foo_context) {
        if (foo_debug) {
            buginf("\n%s FOO: Can't start; "
                   "insufficient system resources",
                   hwidb->hw_short_namestring);
        }
        return;
    }
}

```

4.3.8.5 Displaying Memory Allocation Failures

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

```

Note 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 `nomemory`.

```

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

```

- For major structures that might consume a large amount of memory, consider calling `malloc_named()` 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 now-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.

- `malloc()` does send out and track `malloc` failures, so it is not necessary for the application to mention that unless it is providing additional information.

4.3.8.6 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 `malloc_fast()` function to allocate memory from the 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. Then the pointer whose contents have been freed should be cleared. (`hwidb = NULL`) This stops a pointer from being freed twice, or being dereferenced after it's been freed. This is especially important when the pointer is an element in a structure that isn't being released, or if the pointer is local in scope but may be (re)used later in the code, possibly for a different purpose. Then the code returns a value of `NULL`.

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

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

A better example would be:

```
cdb->cdb_namestring = malloc_fast(sizeof(cdbtype))
if (!cdb->cdb_namestring){
    return;
}

cdb->cdb_description = malloc_fast(strlen(arg) + 1);
if (!cdb->cdb_description) {
    free(cdb->cdb_namestring);
    cdb->cdb_namestring = NULL;
    return;
}
```

4.3.9 Return Memory

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

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

This function frees the size specified in the header of the memory block referenced by the `memory` parameter, so the type of the pointer does not matter. Also, `free()` is NULL-safe, that is, passing a NULL pointer is harmless. This means it is not necessary to write code as:

```
if (ptr) {
    free(ptr)
    ptr = NULL;
}
```

but rather the enclosing `if () {}` should be eliminated as it clutters the code and adds redundant checking:

```
free(ptr);
ptr = NULL;
```

should be the usage.

Actually, let's assume its `*ptr` as a pass-by-reference parameter instead:

```
free(*ptr);
*ptr = NULL;
```

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 *element);
```

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

```
void free(void *element);
```

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 calling the `printf()` function to dump 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. The display process can suspend during a `printf()` call.

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:

```

for (boojum = foo; boojum; boojum = boojumnext) {

    /*
     * Lock the structure since printf may suspend.
     */
    mem_lock(boojum);
    next_save = boojum->next;

    /*
     * Display structure information
     */
    printf("...", boojum->bar);

    .

    /*
     * Unlock structure
     */
    boojumnext = boojum->next;
    free(boojum);

    /*
     * If the node was deleted from the list during printf,
     * abort the operation and inform the user.
     */
    if ((boojumnext == NULL) && (next_save != NULL)) {
        printf("List must have changed...try again\n");
        return;
    }
    boojum = boojumnext;
}

```

And in the thread where the node is deleted, the following must be done:

```

boojum->next = NULL;
free(boojum);

```

You must set 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. `boojum = boojum->next` cannot be in the “for” statement and a local variable `boojumnext` must be used. `boojum = boojum->next` outside a `mem_lock()`/`free()` bracket could fetch a poison value from a memory block that is already freed. This is the main reason that 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 using them as pointers may cause a router crash, or a spurious access error, in the very next iteration of the loop.

Another point to make is that “`boojumnext = boojum->next;`” must be done after the `printf()` call. Doing it before may set `boojumnext` to a memory block that gets freed during `printf()`. This may also result in a router crash/spurious access error, due to the poison value in the memory block that got freed.

One more point—if the “management process” calls `free()` for a memory block that could be an object of a `printf()` somewhere else, the next pointer in the block to be freed must be set to NULL before calling `free()`. For example,

```
clear_all_boojum (...) {
    for (boojum = foo; boojum; boojum = boojumnext) {
        ...
        boojumnext = boojum->next;
        ...
        boojum->next = NULL;
        free(boojum);
        ...
    }
}
```

This is because the data structure freed here may have been `mem_lock()`’ed elsewhere for a `printf()`. Setting `boojum->next` to NULL is necessary so that the “`boojumnext = boojum->next;`” statement inside a `mem_lock()/printf()/free()` bracket would fetch a NULL address, instead of the address of a block that is already freed (and get a poison value, router crash or spurious access).

Note `mem_lock()` provides a locking mechanism for the structure itself, not for its members which could be pointers themselves. This mechanism would not prevent the pointers themselves being deallocated via `free()`, which could lead to problems if `printf()` was printing the fields of the structure and one of them now is junk.

Note The application sets the memory pointer to null once the `refcount` is 0.

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. To set 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);
```

4.3.15 Memchecks Support

Memchecks detects memory corruption errors in C programs as they occur. Memchecks is similar to the IOS Purify tool in that it checks for heap corruption, but it can also detect different types of memory access errors, such as reading or writing freed or unallocated memory, reading and using uninitialized memory, reading and writing from or onto Poison patterns, and illegal function pointers.

For more information on Memchecks, see:

<http://wwwin-ses.cisco.com/memory/memchecks.1.0/index.html>

4.3.16 IOS Memory Scaling

If you need to scale IOS memory, you can increase the alignment of the `.text` and `.data` page starting addresses. You increase this value to a maximum page-size supported by the corresponding MIPS implementation. You can only take advantage of this behavior when there is no available TLB

entry (to map all valid address space) and the platform is MIPS-based. This may result in leaving some memory unused. Dynamic TLB support is not enabled in any production image so this is an issue if the number of TLB entries is still insufficient. See EDCS-520077, "Dynamic TLB Support for IOS," for more information on dynamic TLBs.

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. Chunks are typically used to avoid the memory overhead of `malloc()` when many requests for relatively small memory pieces are needed, or when they are frequently allocated and freed. These chunks are managed by the *chunk manager*. 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 elements allocated from a particular chunk will all be the same size. The maximum size of a dynamic chunk and its siblings is 64K. This size restriction is not for non-dynamic chunks. The size of a chunk block is estimated as follows:
$$((\# \text{ elements} * \text{element size}) + \text{chunk header overhead})$$
- A chunk can grow dynamically, but it cannot grow from the interrupt level (if you need this capability, see the `managed_chunk_create()` reference page). This restriction is because a new chunk sibling requires a memory block allocation (`malloc`), which is not available at the interrupt level.
- When creating a chunk, be careful about the number of chunk elements allocated for each chunk sibling. A chunk sibling is a single memory block allocation (`malloc`) regardless of the number of chunk elements actually in use. When you use the chunk manager, memory is allocated in one large block and the majority of the block will be unused if you allocate too many elements per chunk.
- The chunk creation flag `CHUNK_FLAGS_BIGHEADER` allows for locking of elements via `chunk_lock()`. `CHUNK_FLAGS_BIGHEADER` forces each element to a 4-byte alignment and adds 16 bytes to the size of each element. The 4-byte alignment may create wasted space at the end of each chunk element.

The chunk creation flag `CHUNK_FLAGS_SMALLHEADER` improves performance for chunks with a large number of siblings. When the `SMALLHEADER` option is specified, the chunk manager will add a "prefix" of 4 bytes to each element to store the pointer back to the start of this sibling. Then, when a `chunk_free()` is issued, instead of starting with the master sibling, the chunk manager

can directly reach the correct sibling via the pointer in the prefix. `CHUNK_FLAGS_SMALLHEADER` forces each element to a 4-byte alignment and adds 4 bytes to the size of each element. The 4-byte alignment may create wasted space at the end of each chunk element.

- Use the chunk manager only if the probability of memory corruption is low because the chunk manager has less corruption detection facilities than memory blocks.

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

If you try to create a chunk that is larger than `CHUNK_MEMBLOCK_MAXSIZE` (64k), you will get the following error message:

```
Chunk element size is more than 64k for chunk-name
```

The `flags` parameter can be any combination of the flags listed in Table 4-8.

Table 4-8 **Chunk Pool Flags**

Chunk Pool Flags	Description
<code>CHUNK_FLAGS_DYNAMIC</code>	<p>This flag is typically used if the chunk needs to grow beyond the initial allocation and <code>chunk_malloc()</code> requests are at the process level. When this happens, a new sibling chunk is created using the parameters supplied for the original chunk. After the sibling has been created, all its elements are available for allocation.</p> <p>If the chunk is empty when an interrupt level request occurs, the chunk cannot grow, even if the <code>CHUNK_FLAGS_DYNAMIC</code> flag is set. If this functionality is required, use managed chunks (see the <code>managed_chunk_create()</code> reference page).</p> <p>If the <code>CHUNK_FLAGS_DYNAMIC</code> flag is not specified in the <code>chunk_create()</code> call, the chunk cannot acquire an additional sibling when a <code>chunk_malloc()</code> request occurs and the master chunk is empty.</p> <p>For managed chunks (which perform predepletion expansion via the chunk manager process), this flag allows for both this background and also expansion during a <code>managed_chunk_malloc()</code> function call.</p>
<code>CHUNK_FLAGS_SIBLING</code>	For internal use only. Do not set this flag. Indicates this chunk is a sibling as opposed to the Master Chunk created via <code>chunk_create()</code> .
<code>CHUNK_FLAGS_INDEPENDANT</code>	<p>For each sibling, the element array is allocated separately from the sibling header. This is useful for saving limited space. The chunk header is allocated from heap, <code>MEMPOOL_CLASS_LOCAL</code>.</p> <p>The default is for a single memory allocation to hold both the chunk header and the elements.</p>

Table 4-8 **Chunk Pool Flags (continued)**

Chunk Pool Flags	Description
CHUNK_FLAGS_RESIDENT	<p>Set this flag to reduce thrashing for chunks with many short-duration elements. When all elements for a sibling chunk have been freed, retains the chunk for reallocation. (The default behavior is to return (free) a sibling chunk to the memory manager when all the sibling chunk elements have been freed.)</p>
CHUNK_FLAGS_UNZEROED	<p>Do not zero memory on a <code>chunk_malloc()</code> function call.</p> <p>An important deviation of all Cisco IOS <code>malloc()</code> functions from ANSI standards is that the Cisco IOS <code>malloc()</code> functions all return memory that has been zero filled. While the default behavior of <code>chunk_malloc()</code> is to return zeroed memory, this can be overridden with a <code>chunk_create()</code> call by setting the <code>CHUNK_FLAGS_UNZEROED</code> flag.</p>
CHUNK_FLAGS_BIGHEADER	<p>Allows for locking individual chunk elements using the <code>chunk_lock()</code> function. Each element is enlarged by 16 bytes to hold the “BIGHEADER” prefix. The elements are also 4-byte aligned.</p> <p>The prefix contains a reference count field as well as fields for detection of memory corruption (chunk magic number and <code>allocator_PC</code>) and a pointer back to the start of the chunk (for improved performance).</p> <p>The default is to disallow the <code>chunk_lock()</code> function and not prepend with the 16-byte prefix.</p>
CHUNK_FLAGS_SMALLHEADER	<p>Improves performance for chunks with a large number of siblings by providing faster <code>chunk_free()</code> execution. Each element is enlarged by 4 bytes to hold a pointer to the start of the sibling. Then, when a <code>chunk_free()</code> is issued, instead of starting with the master sibling, the chunk manager can directly reach the correct sibling via the pointer in the prefix.</p> <p>The default is to not prepend the 4-byte prefix, lengthening the execution path for <code>chunk_create()</code> calls.</p> <p>Note Some developers have switched to small headers (even for very small chunks) because significant chunks of CPU time were being expended while freeing chunks on highly scaled systems. For example, you can switch to <code>CHUNK_FLAGS_SMALLHEADER</code> to avoid having CPUHOG problems on <code>chunk_free()</code> if the chunk is dynamic and has a large number of elements.</p>
CHUNK_FLAGS_MANAGED	<p><code>CHUNK_FLAGS_MANAGED</code> must be set in the <code>managed_chunk_create()</code> function call; do not set this flag under other conditions.</p> <p><code>CHUNK_FLAGS_MANAGED</code> indicates the chunk is a managed chunk. The caller must use the set of <code>managed_chunk_xxxx()</code> calls and then the chunk manager will perform predepletion expansion for this chunk.</p> <p>Managed chunks are useful when an application (a) needs to acquire chunk elements at the interrupt level or (b) desires that the expansion be moved from inline in this process and be performed by the chunk manager process. (a) is the most common requirement; (b) is an unlikely situation.</p> <p>Managed chunks are new in release 12.2.</p>

Table 4-8 **Chunk Pool Flags (continued)**

Chunk Pool Flags	Description
CHUNK_FLAGS_NONDATA	<p>Indicates the chunk elements will reside on an external memory device. Management of the external memory device is external to the chunk manager and is defined by routines defined at <code>alloc_exmem_p</code> and <code>free_exmem_p</code>. This flag forces <code>CHUNK_FLAGS_INDEPENDANT</code>.</p> <p>The default is for the chunk elements to be allocated with the chunk header in the memory type specified in the <code>chunk_create()</code> default from <code>MEMPOOL_CLASS_LOCAL</code>.</p> <p>External data chunks were first introduced in 12.0S.</p>
CHUNK_FLAGS_TRANSIENT	<p>The Transient Memory Allocation feature provides a transient mempool to serve the memory allocation requests that are transient in behavior. If a <code>chunktype</code> is transient in nature, then the <code>CHUNK_FLAGS_TRANSIENT</code> flag should be passed. All the siblings of this chunk would also be allocated from the transient memory pool. For more information about this feature, which is available on a limited number of platforms only, see section 4.6, “Transient Memory Allocation (New in 12.2S).”</p>
CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF	<p>The application programmer uses the flag called <code>CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF</code> to set the interrupt level while manipulating certain chunks. If the <code>CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF</code> flag is TRUE, then there is no protection against interrupts. If the <code>CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF</code> flag is FALSE, then there is protection against interrupts. Protection against interrupts decreases system performance, so the application programmer would want to use this flag to improve system performance. By default, this flag is FALSE.</p> <p>Application programmers should use this flag with caution. The application programmer who plans to use the <code>CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF</code> flag should ensure that the interrupts never occur while the flag is TRUE. An interrupt that occurs during chunk manipulation may cause memory or data corruption, which leads to a system crash. (An example of this is CSCsi73899.)</p> <p>The <code>CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF</code> flag is new in Cisco IOS Release 12.2SR(Autobahn76).</p>
CHUNK_FLAGS_NONLAZY	<p>This flag indicates that this chunk is nonlazy; that is, delayed allocation of chunk elements is not performed for the chunk when this flag is set. The <code>CHUNK_FLAGS_NONLAZY</code> flag is first integrated in 12.4(20)T.</p>
CHUNK_FLAGS_NOHEADER_FAST	<p>This flag makes the execution of <code>chunk_free()</code> faster by introducing an AVL tree in the chunk and its siblings. The AVL tree is formed based on the chunk data address. This tree is searched during the execution of the <code>chunk_free()</code> function to identify the chunk where the chunk element belongs. This approach avoids the overhead of introducing a header for each chunk element. This flag introduces an AVL info structure along with every chunk created. The size of this info structure for a 32 bit architecture is 20 bytes in a chunk root and 16 bytes in case of a chunk sibling. Note that this flag is not valid for small header, big header, and non-dynamic chunks.</p>

These flags are defined by the chunk manager in `chunk.h` as follows:

```
#define CHUNK_FLAGS_NONE          0x00000000
#define CHUNK_FLAGS_DYNAMIC        0x00000001
#define CHUNK_FLAGS_SIBLING         0x00000002
#define CHUNK_FLAGS_INDEPENDANT    0x00000004
#define CHUNK_FLAGS_RESIDENT        0x00000008
#define CHUNK_FLAGS_UNZEROED        0x00000010
#define CHUNK_FLAGS_BIGHEADER       0x00000020
#define CHUNK_FLAGS_SMALLHEADER     0x00000040
#define CHUNK_FLAGS_MANAGED         0x00000080
#define CHUNK_FLAGS_NONDATA         0x00000100
#define CHUNK_FLAGS_TRANSIENT       0x00000200
...
#define CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF 0x00001000
#define CHUNK_FLAGS_NONLAZY         0x00002000
#define CHUNK_FLAGS_NOHEADER_FAST   0x00010000
```

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 chunk siblings 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` number of elements, and each element size is `sizeof(rdbtype)`.

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

Note The lazy chunk allocation feature is present in 12.4(16.14)T1. With this feature, `chunk_create()` will create only the chunk header. The chunk elements will only be created at the time of the first `chunk_malloc()`.

4.4.3.2 Dynamic Chunk Sibling Creation and Destroy

One of the powerful features of chunks is their ability to grow and shrink dynamically over time. When `chunk_create()` is called, the master chunk is allocated which will hold *maximum* number of elements, each of *size* bytes.

If the `CHUNK_FLAGS_DYNAMIC` flag is set in the `chunk_create()` function call, the following will occur:

- 1 When all elements are allocated via `chunk_malloc()`, a new “sibling” chunk will be acquired. The sibling has the same characteristics as the master, with the same *maximum*, *size*, and *alignment* parameters.
- 2 When elements are returned via `chunk_free()`, and the sibling chunk now has no elements in use, this sibling will be returned to the memory management system via `free()`.

If the `CHUNK_FLAGS_DYNAMIC` flag is not set in the `chunk_create()` function call, no dynamic sibling creation will be performed.

4.4.3.2.1 A Description of Chunk Sibling Chaining

The following is a description of chunk sibling chaining:

Master chunk ---> Sib 500 ---> Sib 499 ---> Sib1

The reason for this is that the newly created chunk sibling would be appended right after the master chunk so that the next `chunk_malloc()` could easily get the chunk element from this sibling rather than walking through the entire list of siblings.

When a sibling chunk is depleted by a `chunk_malloc()` (that is, the last free element was given out), it is removed from its current position at the front part of the sibling linked list and appended to the end of the list. This is so that the next `chunk_malloc()` can find a free chunk element from the front part of the sibling linked list without going through this empty sibling chunk.

When a sibling chunk is fully populated after `chunk_free()` (that is, all elements of this sibling have been returned and are again available), this sibling is removed from its current position in the rear part of the sibling linked list. It is appended right after `head_chunk` if its `next_sibling` is empty or after `head_chunk->next_sibling` if `next_sibling` is not empty. The next `chunk_malloc()` can then find a free chunk element from the front part of the sibling linked list.

4.4.3.3 External Memory

For external memory, `exfree.c` handles memory (like `free.c`) with the exception that the memory is not accessible by the handler in `exfree`. For example:

- The memory handle is running on the RP but the memory is on the LC.
- The memory being handled is some special memory that is not CPU memory (NONDATA), such as ASIC tables.

Of course, `free.c` does not support these cases because it needs to access the memory to be able to manage it. Therefore, the ability of chunk elements to run on the top of `exfree` by using the `CHUNK_FLAGS_NONDATA` flag was added, similar to the way the chunk elements run on the top of `free`.

Note The Memory Leak Detector in IOS (also commonly known as the Garbage Detector) does not work on externally managed memory; for example, `sys/os/exfree.c`.

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.

Note Currently the standard Cisco IOS kernel only supports internal memory. However a prototype for external memory is provided in the GSR code base. Both `malloc()` and `chunk_malloc()` support is available. Please refer to the 12.0ST and later code trains.

If the chunk was created with locking and the reference count is greater than 1, the reference count is decremented and `chunk_free()` returns without further action. Chunk element locking is implemented within the `chunk_create()` function with the `CHUNK_FLAGS_BIGHEADER` flag set.

If the chunk was created with faster free processing and the `*chunk` argument is `NULL`, the header prepending the chunk element has a pointer back to this chunk sibling and the entire list of chunk siblings need not be searched.

Without faster free processing, the list of siblings must be searched to find the sibling containing this element. Faster chunk element free is implemented within the `chunk_create()` function with the `CHUNK_FLAGS_SMALLHEADER` or `CHUNK_FLAGS_BIGHEADER` flag set.

Note In 11.3 and below only, the `CHUNK_FLAGS_BIGHEADER` implementation was supported and the flag name was `CHUNK_FLAGS_DATAHEADER`.

Before returning the chunk element, it is filled with the poison pattern. For performance reasons, only a limited number of bytes is filled with the poison pattern (`MAX_CHUNK_POISON`). If an application needs more poison bytes, it must be done before calling `chunk_free()`.

If the chunk was created with dynamic sibling return and this is the last element in the sibling chunk, the memory for that sibling will be returned to the appropriate memory pool. Dynamic sibling return is enabled within the `chunk_create()` function by NOT setting the `CHUNK_FLAGS_RESIDENT` flag.

If elements are returned at the interrupt level, use managed chunks instead, defined in the [managed_chunk_create\(\)](#), [managed_chunk_malloc\(\)](#), [managed_chunk_free\(\)](#), and [managed_chunk_destroy\(\)](#) reference pages.

If the last element in the chunk sibling is returned at interrupt level, memory leaks will occur because the sibling is removed from the queue of siblings for this chunk; however, the `free()` function will fail.

Possible error messages:

`%SYS-2-CHUNKNOROOT: Root chunk need to be specified for [hex element address]`

This message is rate limited to once every 10 seconds.

`%SYS-2-CHUNKBADMAGIC: Bad magic number in chunk header, chunk [hex supplied chunk pool address] data [hex supplied element address] chunkmagic [hex data at location of chunk magic for element] chunk_freemagic [hex contents of freemagic field within element]`

This message is rate limited to once every 10 seconds. It is only issued if the chunk was established with BIGHEADERS.

`%SYS-2-CHUNKBADREFCOUNT: Bad chunk reference count, chunk [hex supplied chunk pool address] data [hex supplied element address]`

This message is rate limited to once every 10 seconds. It is only issued if the chunk was established with BIGHEADERS.

4.4.4.1 Example: Allocate a Memory Chunk

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

```
rdbype *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_BIGHEADER` 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(void *data);
```

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.

If the chunk is not empty, it returns FALSE and you should check the return code.

If the chunk has a sibling, you'll get the rate-limited message and it returns FALSE. The rate-limited message is:

```
%SYS-2-CHUNKSIBLINGS Attempted to destroy chunk with sibling
```

4.5 Managed Chunks (New in Release 12.2)

Managed chunks allow for “pre-depletion expansion.” If you remember with regular chunks, you can set the `CHUNK_FLAG_DYNAMIC` and when there are no more elements, the chunk manager code will automatically go out and do another `malloc()` and create a sibling chunk. The problem is, if you run out and the call is from the interrupt level, you can't do a `malloc()`. Therefore, managed chunks will do the sibling creation *before* you totally run out.

4.5.1 Setting Up a Managed Chunk

Setting up a managed chunk requires the following:

- determining the element size
- determining the appropriate number of elements per sibling
- selecting the “low water mark” threshold below which the expansion is requested. Remember that the expansion takes place at the process level, so even though this is a PRIO_CRITICAL process, consider delays until the sibling is established.
- making sure to set the flag CHUNK_FLAGS_MANAGED when you call [managed_chunk_create\(\)](#) See Table 4-8.
- selecting additional options. See Table 4-8.

4.5.2 How Managed Chunks Work

A new process is added into the mix. The “Chunk Manager” process, `managed_chunk_process()` is unconditionally spawned at system initialization. It watches a new managed queue - `managed_chunk_queue()`.

For each `managed_chunk_malloc()`, the number of available elements is compared to the “expansion_threshold” provided when calling `managed_chunk_create()`. If the number of available elements is less than “expansion_threshold”, the managed chunk is enqueued to `managed_chunk_queue()` to be expanded at process level.

Expansion is no different from regular chunks—`malloc()` a new sibling and place it into the chunk chain.

For each `managed_chunk_free()`, similar logic takes place. The sibling will only be returned (yes, by enqueue to `managed_chunk_queue()` for later return at process level) if the following is true:

- All elements in this sibling are available.
- The total number of available elements is $> 2 * \text{expansion_threshold}$.
- The total number of available elements is greater than the maximum number of elements per sibling.

4.5.3 Create a Managed Chunk

Managed chunks are created when the system is initialized by calling the `managed_chunk_create()` function. Creating the managed chunk registers it automatically for the flags specified in the `flags` parameter, and the chunk immediately becomes available for operations by the `malloc()` and `free()` functions.

```
extern chunk_type *managed_chunk_create (ulong size,
                                         ulong maximum,
                                         ulong flags,
                                         mempool *mempool,
                                         ulong alignment,
                                         char *name,
                                         ulong expansion_threshold);
```

4.6 Transient Memory Allocation (New in 12.2S)

Fragmentation is a common occurrence in IOS. It can be attributed to a combination of the allocation and de-allocation patterns of memory and the excessive allocation of small-sized blocks.

Transient memory is memory that is allocated and de-allocated within a short period of time. In contrast, static-allocated memory is memory that is expected to remain allocated for a significant period of time. For example, transient memory allocations are not expected to outlive the logical flow of processing for an event such as network or timer events. Static memory allocations can be expected to outlive the logical flow of processing for an event; the logical flow could even span across process boundaries.

As an example of transient memory, when the SNMP agent receives an SNMP packet, it parses the packet into internal SNMP structures (transient). These SNMP structures are then passed to the MIB routine for further processing. If the request results in the creation of a new row, then additional data structures are allocated by the MIB to represent the row (static). Finally, when the MIB routine finishes processing, the result is again packed into internal SNMP data structures (transient), which are then encoded into an outgoing SNMP packet. In the context of this example, the data structures marked “transient” are allocated and freed during the processing of the SNMP packet, while the data structures marked “static” are expected to remain allocated until some other event causes their de-allocation.

Another example would be BGP. This routing protocol does many transient and static allocations due to routing updates from the neighboring routers.

Most of IOS code exhibits such a memory allocation pattern, where there is an interleaving of transient and static memory allocations. Once transient memory is freed, the blocks cannot be coalesced because of the static memory sitting between them, thus creating fragmentation.

To address this problem, the Transient Memory Allocation feature has been developed to allocate all transient memory in a separate memory address space (separate region), so that there won’t be any interleaving of static and transient memory blocks.

This feature has been developed for initial release on the 7200, 7300, and 7500 platforms, with more platforms pending further testing.

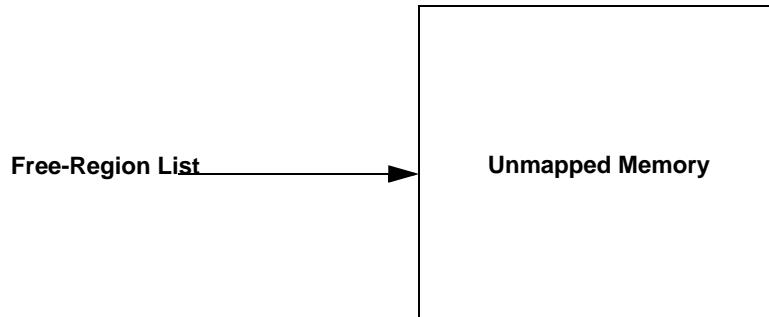
4.6.1 Dynamic Region Manager

A new dynamic region manager has been added to maintain a freeregionlist and provide memory (as regions) to memory pools as and when needed. A new transient memory pool has also been added to serve the memory allocation requests that are transient in behavior.

After creating Text, Data, and BSS regions, the dynamic region manager checks the remaining memory. If there is at least 64 MB for heap, then the transient memory feature can be enabled.

If a platform needs to enable the Transient Memory Allocation feature, `region_init()` is called by `platform_memory_init()` during platform memory initialization and initializes the free-region list (freeregionlist). `region_init()` is passed the start of the unmapped memory and the size of the memory. The size is checked to establish if there is sufficient memory for the free-region list. If not, then the whole memory is given to heap memory. If there is sufficient memory, 16 MB is reserved for heap memory and the remainder is reserved as a free region.

Initially, the free-region list will have one region in it, as displayed in Figure 4-5:

Figure 4-5 Layout of the Free-Region List after System Initialization

The regionlist is as follows in Figure 4-6:

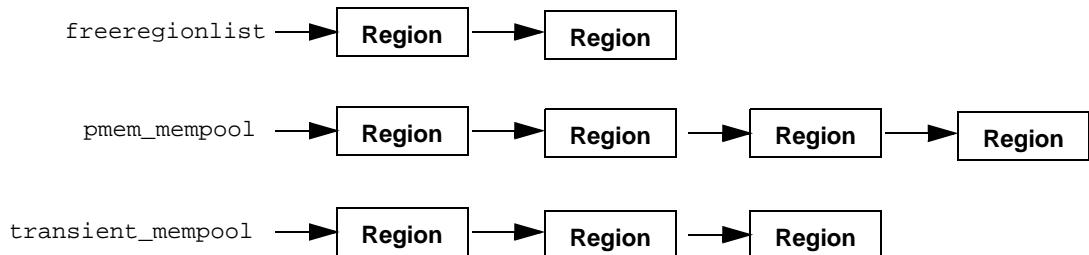
Figure 4-6 Layout of the Region List after System Initialization

When heap (16 MB) becomes depleted, new regions will be supplied from the free-region list.

4.6.2 Static and Transient Memory Pools

During router initialization, the processor memory pool, `pmem_mempool`, is created at a region size of 16 MB. The `malloc()` function services static memory allocation requests from `pmem_mempool`. For transient memory allocation requests, a new API function, `tm_malloc()`, has been developed. The first time `tm_malloc()` is called, a transient memory pool, `transient_mempool`, is created at a region size of 16 MB. As a result, transient blocks are restricted to a separate memory pool, thereby solving the fragmentation issue. The processor memory pool, `pmem_mempool`, serves as a fallback to the transient memory pool, `transient_mempool`.

The number of regions associated with a memory pool depends upon the usage of memory. Here's a potential usage map, Figure 4-7:

Figure 4-7 Regions Associated with the Free-Region List and Static and Transient Memory Pools

In summary, the `pmem_mempool` services `malloc()` calls, the `transient_mempool` services `tm_malloc()` calls, and `free()` adds the freed block to the appropriate memory pool and checks if the corresponding region can be added to the free-region list.

4.6.2.1 Managing Memory for Memory Pools and the Free-Region List

All the `malloc()` requests are served from `pmem_mempool`. Once this region, initially set to 16 MB, is emptied, the next region is created based on the requested size. For example, if `memsize` is the request through `malloc()`, then the size to be allocated is calculated as follows:

```
* MIN_PMEM_POOL_SIZE is 16MB. While creating new regions, we create a region
* of at least 16MB size.

* Add the overhead needed for blocktype.
if (memsize < MIN_PMEM_POOL_SIZE) {
    memsize = MIN_PMEM_POOL_SIZE;
}
* Create a region from the freeregionlist with this new memsize.
* Add the new region to the mempool.
* Do a malloc_block () from the mempool now.
```

The first call to `tm_malloc()` adds a 16MB region to the transient mempool. Once this 16MB becomes depleted, a new region will be allocated from the free-region list. The way the new region is allocated and added is an internal implementation that also might change in the future.

4.6.2.2 Freeing Memory to Add to freeregionlist

While calling `free()` to release a block, the region in which the block is present will be checked in order to see if it has any INUSE blocks in it. If not, then the whole region will be removed from the memory pool and will be added to the free-region list. While adding a region to the free-region list, care will be taken so that the region will coalesce with its neighboring contiguous regions, the same method as used for a blocktype in a memory pool.

4.6.3 Transient Memory API Functions

The Transient Memory Allocation feature adds the following new functions:

- `region_init()`
- `tm_malloc()`
- `tm_malloc_aligned()`

4.6.3.1 Initializing the Free-Region List

If a platform needs to enable the Transient Memory Allocation feature, `region_init()` is called by `platform_memory_init()` during platform memory initialization and initializes the free-region list. `region_init()` is passed the start of the unmapped memory and the size of the memory. The size is checked to establish if there is sufficient memory for the free-region list. If not, then the whole memory is given to heap memory. If there is sufficient memory, 16 MB is reserved for heap memory and the remainder is reserved as a free region.

```
boolean region_init(regiontype *region, void * heap_start, uint size)
```

4.6.3.2 Allocating Memory from the Transient Memory Pool

Use the `tm_malloc()` function to allocate a new block that is bigger than the requested `size` from the transient memory pool, `transient_mempool`.

```
void *tm_malloc(size_t size)
{
```

```

mempool_class class = MEMPOOL_CLASS_TRANSIENT;
SAVE_CALLER();

if (!mempool_find_by_class_inline(MEMPOOL_CLASS_TRANSIENT)) {
    class = tm_mempool_create(size) ?
        MEMPOOL_CLASS_TRANSIENT : MEMPOOL_CLASS_LOCAL;
}

return (malloc_aligned_inline(class, caller(), size, 0, FALSE, 0));
}

```

4.6.3.3 Allocating Aligned Memory from the Transient Memory Pool

Use the `tm_malloc_aligned()` function to allocate a new aligned block that is bigger than the requested `size` from the transient memory pool, `transient_mempool`.

```

void *tm_malloc_aligned(uint size, uint align)
{
    mempool_class class = MEMPOOL_CLASS_TRANSIENT;
    SAVE_CALLER();

    if (!mempool_find_by_class_inline(MEMPOOL_CLASS_TRANSIENT)) {
        class = tm_mempool_create(size) ?
            MEMPOOL_CLASS_TRANSIENT : MEMPOOL_CLASS_LOCAL;
    }

    return (malloc_aligned_inline(class, caller(), size, align, FALSE, 0));
}

```

4.6.4 Transient Chunks

A new flag, `CHUNK_FLAGS_TRANSIENT`, has been added. To indicate if a `chunk_type` is transient, pass the `CHUNK_FLAGS_TRANSIENT` flag along with the other flags. All the siblings for this chunk will be allocated from the transient memory pool, `transient_mempool`. The new flag is defined in `chunk.h`:

```
#define CHUNK_FLAGS_TRANSIENT 0x00000200
```

For more information about using chunks, see 4.4.3 “Create a Memory Chunk.”

4.6.5 Transient Memory and CLI Output

There are two commands that are of relevance to the use of transient memory. They are:

- **show region**
- **show memory**

4.6.5.1 show region

The output of the **show region** EXEC command shows the locations and sizes of the memory regions in bytes. With transient memory enabled, the command output now includes the regions assigned to all memory pools and the freeregionlist.

The following is an example output from the **show region** EXEC command:

```
Region Manager:
```

Start	End	Size(b)	Class	Media	Name
0x07000000	0x07FFFFFF	16777216	Iomem	R/W	iomem
0x60000000	0x66FFFFFF	117440512	Local	R/W	main
0x60008C40	0x61B16A67	28368424	IText	R/O	main:text
0x61B18000	0x62AF027F	16614016	IData	R/W	main:data
0x62AF0280	0x62E5065F	3539936	IBss	R/W	main:bss
0x62E50660	0x62FDEE7B	1632284	Local	R/W	main:saved-data
0x62FDEE7C	0x63FDEE7B	16777216	Local	R/W	main:heap
0x77000000	0x77FFFFFF	16777216	Iomem	R/W	iomem:(iomem_cwt)
0x80000000	0x86FFFFFF	117440512	Local	R/W	main:(main_k0)
0xA0000000	0xA6FFFFFF	117440512	Local	R/W	main:(main_k1)

Free Region Manager:

Start	End	Size(b)	Class	Media	Name
0x63FDEED0	0x66FFFFFF	50467120	Local	R/W	heap

4.6.5.2 show memory

The output of the **show memory** EXEC command displays statistics about memory, including memory-free pool statistics. With the addition of the transient memory pool, you can view memory pool statistics just like those of the processor memory pool.

The following is an example output from the **show memory** EXEC command:

```
ios102#sh mem
          Head   Total(b)    Used(b)     Free(b)    Lowest(b)   Largest(b)
Processor 62CE4C00  33554432  22146832  11407600  11277904  10765276
      I/O 20000000  33554584  652520    32902064  32902064  32885400
    I/O-2 E000000  33554448  2380656   31173792  31173792  31173752
Transient 64CE4CA8  16777232  3280512   13496720  13496720  13496672
```

The command **show memory transient** is also available, which limits the output of memory pool statistics to the transient memory pool only.

4.7 Memory Management Commands

Various **show** commands are useful for checking aspects of memory, and a number of **memory** commands can be used to control memory allocation and validation.

Note The **sh mem 0x addr** command is now an internal command that is not available unless you enter the **service internal** config command (New in 12.3, 12.4, 12.2S, and 12.SX via CSCeg23300).

See Table 4-9 for some **show** commands and **memory** commands that are useful to developers for checking memory.

Refer to the following sources for other useful commands and information related to memory management issues:

- *Buffer Overflow: Detection and Correction of Redzone Corruption:*

http://www.cisco.com/en/US/docs/ios/12_3t/12_3t7/feature/gtbufflo.html

- Cisco IOS Configuration Fundamentals Command Reference, Release 12.4: **show memory** commands:
http://www.cisco.com/en/US/docs/ios/fundamentals/command/reference/cf_book.html
- *Embedded Resource Manager: (Added in 12.3T and later releases)* Includes a feature to configure thresholding for memory usage accounting on a per user local, per user global, or system-wide basis, configured using resource policy commands applied to Memory Resource Users:
http://www.cisco.com/en/US/docs/ios/netmgmt/configuration/guide/nm_erm_resource_ps6922_TSD_Products_Configuration_Chapter.html
- *Memory Leak Detector Product Feature Guide*, also referred to as Garbage Detection in IOS: *(Added in 12.2S, 12.3T, and later releases)* Identifies memory leaks and allows the memory to be released and reused:
http://www.cisco.com/en/US/docs/ios/fundamentals/configuration/guide/cf_mem-leak-detect_ps6350_TSD_Products_Configuration_Guide_Chapter.html

Table 4-9 lists some helpful memory-related commands, with references for command syntax, examples, and more information.

Table 4-9 Memory-Related Commands

Command	Description
memory check-interval	Hidden command to set the frequency that the <code>checkheaps</code> process runs to validate block headers. More information on this command can be found in section 4.7.3, “Memory Validation Commands”.
memory malloc-list-use-malloc	Hidden command to instruct the <code>malloc_lite()</code> API to use normal <code>malloc()</code> infrastructure, instead of the <code>malloc_lite</code> infrastructure. More information on this command can be found in section 4.7.4, “Malloc-Lite Memory Commands”.
memory statistics history table	<i>Added in 12.3T:</i> Sets the log time in hours for memory statistics history, and shows memory consumption history as a statistics table.
show memory statistics history table	More information on this command can be found in the following Cisco IOS documentation: <i>Cisco IOS Configuration Fundamentals Command Reference (12.4):</i> http://www.cisco.com/en/US/docs/ios/fundamentals/command/reference/cf_book.html <i>Embedded Resource Manager (ERM) Feature Guide:</i> Managing Memory Usage History: http://www.cisco.com/en/US/docs/ios/netmgmt/configuration/guide/nm_erm_resource_ps6350_TSD_Products_Configuration_Guide_Chapter.html#wp1054827 Verifying ERM Operations: http://www.cisco.com/en/US/docs/ios/netmgmt/configuration/guide/nm_erm_resource_ps6441_TSD_Products_Configuration_Guide_Chapter.html#wp1062676
memory validate-checksum	Hidden command to set the frequency that the <code>checkheaps</code> process runs to check for memory corruption in the text segment. More information on this command can be found in section 4.7.3, “Memory Validation Commands”.
memory try-malloc-lite	Hidden command to instruct the <code>malloc()</code> API to try <code>malloc_lite()</code> for small amounts of memory. More information on this command can be found in section 4.7.4, “Malloc-Lite Memory Commands”.

Memory Management Commands

Table 4-9 Memory-Related Commands (continued)

Command	Description
set memory debug incremental starting-time	<i>Added in 12.3T and 12.2S:</i> Debug command to set a start time from which to begin monitoring memory allocation and leak detection, then display information starting at that time about memory allocations (allocations), leaks detected (leaks), and the incremental start time and current elapsed time (status). More information on this command and its other options can be found in the <i>Memory Leak Detector Feature Guide</i> : http://www.cisco.com/en/US/docs/ios/12_3t/12_3t7/feature/guide/gtmleakd.html
show chunk	Hidden command that shows the locations and sizes, in bytes, of the memory chunks.
show mem big	Hidden command that displays the largest 25 memory blocks on the router. The output just stops with the 25th-largest block even if there are more similar-sized blocks.
show memory	Shows the locations and sizes, in bytes, of all memory blocks. More information on this command can be found in section 4.7.2.1, “show memory Output”.
show memory allocating-process [totals]	Shows the statistics on the allocated memory with the corresponding allocating process.
show memory dead	Shows the locations and sizes, in bytes, of dead memory. More information on this command can be found in section 4.7.2.3, “show memory dead Output”.
show memory debug leaks [chunks]	<i>Added in 12.3T and 12.2S:</i> Debugging command to display information about memory leaks. Adding the chunks keyword includes information about chunks. More information on this command and its other options can be found in the <i>Memory Leak Detector Feature Guide</i> (internally referred to as the Garbage Detector): http://www.cisco.com/en/US/docs/ios/12_3t/12_3t7/feature/guide/gtmleakd.html See also Chapter 20, “Debugging and Error Logging”, and additional references for Garbage Detection in section 20.7, “Links to Other Debugging Documentation”.
show memory debug references	Displays the debug information on references.
show memory ecc	Displays the single-bit Error Code Correction (ECC) error logset data.
show memory failures alloc	Shows the last 10 memory allocation failures. More information on this command can be found in section 4.3.8.5, “Displaying Memory Allocation Failures”.
show memory fast [allocating-process [totals] dead [totals] free [totals]]	Shows the fast memory details for the router.
show memory fragment	Shows the block details of the fragmented free blocks and allocates the blocks that are physically just before or just after the blocks on the free list.
show memory free	Shows the locations and sizes, in bytes, of memory blocks which are not currently in use. More information on this command can be found in section 4.7.2.2, “show memory free Output”.
show memory multibus	Displays the statistics on multibus memory, including the memory-free pool statistics.
show memory pci	Displays the statistics on the Peripheral Component Interconnect (PCI) memory.
show memory processor	Displays the statistics on the router processor memory.
show memory scan	Monitors the number and type of parity (memory) errors on the system.

Table 4-9 Memory-Related Commands (continued)

Command	Description
show memory transient [allocating-process [totals] dead [totals] fragment [detail] free [totals] statistics [history]]	Displays the statistics on transient memory.,
show processes memory	Displays the sum of allocated memory by process ID as found in the PID structure. More information on this command can be found in section 4.7.2.14, “show processes memory Output”.
show region	Shows the locations and sizes, in bytes, of the memory regions. More information on this command can be found in section 4.7.1.2, “Display Memory for Each Region”.
show version	Shows total memory on your system. More information on this command can be found in section 4.7.1.1, “Display Total System Memory”.

4.7.1 Determine Amount of Memory Available

4.7.1.1 Display Total System Memory

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 126976 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 IOS Software, 3600 Software (C3640-I-M), Experimental Version 12.3. . .
Copyright (c) 1986-2004 by Cisco Systems, Inc.
Compiled Fri 30-Apr-04 00:02 by haj

ROM: System Bootstrap, Version 11.1(19)AA, EARLY DEPLOYMENT RELEASE SOFTWARE
(fcl)
ROM: 3600 Software (C3640-I-M), Experimental Version 12.1. . .

Router uptime is 3 weeks, 11 hours, 32 minutes
System returned to ROM by abort at PC 0x60481AE4 at 15:52:27 UTC Thu Apr 29
2004
System restarted at 07:26:50 UTC Fri Apr 30 2004
System image file is "tftp://172.19.192.254/haj/c3640-i-mz.parser"

Cisco 3640 (R4700) processor (revision 0x00) with 126976K/4096K bytes of
memory.
Processor board ID 11908952
R4700 CPU at 100MHz, Implementation 33, Rev 1.0
4 Ethernet interfaces
4 Serial interfaces
DRAM configuration is 64 bits wide with parity disabled.
```

```

125K bytes of NVRAM.
32768K bytes of processor board System flash (Read/Write)
16384K bytes of processor board PCMCIA Slot0 flash (Read/Write)
20480K bytes of processor board PCMCIA Slot1 flash (Read/Write)

Configuration register is 0x0

```

4.7.1.2 Display Memory for Each Region

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	0x033FFFFFF	4194304	Flash	R/O	flash
0x06000000	0x063FFFFFF	4194304	Iomem	R/W	iomem

4.7.2 Memory Display Commands

A number of the memory display commands are detailed in the following subsections.

4.7.2.1 show memory Output

The **show memory** command displays each block of memory in order of ascending memory address as in:

```

. . .
Address Bytes      Prev      Next      Ref PrevF      NextF      Alloc PC what
618F3C00 0002876608 00000000 61BB20E8 000 0          6201A850 604A0C3C
(coalesced)
61BB20E8 0000020004 618F3C00 61BB6F34 001 ----- ----- 6058ADCC Managed
. .
61BB6F34 0000001504 61BB20E8 61BB753C 001 ----- ----- 60550F40 List
Elements
. .

```

Because this display is in order of blocks within memory, the *Next* column is the next line in the display and the *Prev* column is the previous line in the display.

When the *Ref* column is zero, as with the first entry, the block is available and the *PrevF* and *NextF* columns will indicate the Freelist chain. The freelist chains can be viewed with the **show memory free** command.

The next two entries are allocated with the *Ref* column greater than zero and the *PrevF* and *NextF* columns excluded.

Table 4-10 displays information about each block in memory.

Table 4-10 show memory Column Description

Column Name	Description
Address	Starting address of the header for this memory block.
Bytes	Decimal number of bytes for this block, excluding the block header.
Prev	Address of the previous block in this region.
Next	Address of the next block in this region.
Ref	Number of references to this block. 0 indicates the block is available and on a free list. 1 indicates this block is in use. A number greater than 1 indicates this block is allocated and also has additional references, via the mem_lock() function.
PrevF	If this block is available, the address of the previous block on the freelist.
NextF	If this block is available, the address of the next block on the freelist.
Alloc PC	Address of the call to the malloc() function.
what	The name of the process which allocated this block or a name given to this block via the malloc_named() suite of functions.

4.7.2.1.1 The blocktype Header and **previous

Regarding offsetting and how **previous does *not* point to the beginning to the previous block, instead it points to *next in the previous block, here is an explanation of the memory blocks:

```
R100#sh mem | i Prev|02407070
  Address      Bytes      Prev      Next  Ref      PrevF      NextF Alloc PC  what
+-02404928 0000010000 024036A8 02407070 001  ----- 011EF3DC LSPV: DS Out...
+-02407070 0000000328 02404928 024071F0 001  ----- 005EA1CC Chain Cache No
+--024071F0 0000000064 02407070 02407268 001  ----- 00B184CC OSPF-1 Router
|| R100#
|| R100#
|| R100#
|| R100#
|| R100#
R100#sh mem 0x02404928 0x02404943
|| +>02404920: AB1234CD 00760000      +.4M.v...
|| 02404930: 03689868 0201C4C0 011EF3DC 02407070 .h.h..D@..s\.@pp
|| 02404940: 024036BC      *next      .@6<
||          * **previous           | |
||          | ^                   |
||          | |
R100#
R100#sh mem 0x02407070 0x0240708b      | +-----+
|| +-----+                               |
|| V                                |
+->02407070: AB1234CD 00340000 03302BE0 0246FC9C +.4M.4...0+.F|
|| 02407080: 005EA1CC 024071F0 0240493C      .^!L.@qp.@I<
||          *next      **previous -----+
||          | ^                   |
||          | +-----+             |
||          +-----+             |
R100#
R100#sh mem 0x024071F0 0x0240720B      | +-----+
|| +-----+                               |
|| V                                |
+->024071F0: AB1234CD 007B0000 03540178 02B733D4 +.4M.{...T.x.73T
|| 02407200: 00B184CC 02407268 02407084      .1.L.@rh.@p.
||          *next      **previous -----+
||          | ^                   |
||          | +-----+             |
||          +-----+             |
R100#
```

Note The size of `blocktype_` was once a standard across IOS branches that was set at 40. This is no longer the case; refer to the code base of your specific branch for details. One method that you can use to identify size of the `blocktype` would be to do a “`(cisco-6.4.2-r4k-gdb)p sizeof(blocktype)`” by using the sunfile or by calculating using the `(blocktype_)` definition in a specific branch.

For additional information, see also “Chaining the Blocks Together” and “Cisco IOS Memory Structures” at the following URLs respectively:

<http://wwwin-engd.cisco.com/common/courses/mem/ID-8.html#pgfId-25993>

http://wwwin-engd.cisco.com/common/courses/mem/ID-49.html#0_pgfId-6617

4.7.2.2 show memory free Output

The **show memory free** command runs through each freelist, displaying blocks of free memory in the order that they reside in the appropriate freelist.

For **show memory free**, we are going through the freelists so entry 1 NextF is entry 2; entry 2 nextF is entry 3.

```
.
.
.
Address Bytes Prev Next Ref PrevF NextF Alloc PC what
24 Free list 1
61E63D7C 0000000048 61E63D4C 61E63DD8 000 0 61E9CF54 608042F8 CDP
Protocol
61E9CF54 0000000088 61E9CE6C 61E9CFD8 000 61E63D7C 61E9CDFC 60483F7C Exec
61E9CDFC 0000000048 61E9CD24 61E9CE58 000 61E9CF54 61E973FC 60483F7C Exec
61E973FC 0000000048 61E972F8 61E97458 000 61E9CDFC 61FC19B8 60483F7C Exec
61FC19B8 0000000048 61FC18E0 61FC1A14 000 61E973FC 61FC1C6C 60483F7C Exec
61FC1C6C 0000000048 61FC1C38 61FC1CC8 000 61FC19B8 61E9CA24 60483F7C Exec
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

Since these are only available blocks, the `Ref` column will always be 0.

Because this display is in order of blocks on a freelist, the `NextF` column is the next line in the display and the `PrevF` column is the previous line in the display.

4.7.2.3 show memory dead Output

The **show memory dead** command displays details about the dead memory blocks in the system.

```
.
.
.
Address Bytes Prev Next Ref PrevF NextF Alloc PC what
61BD6220 0000000084 61BD51F4 61BD629C 001 ----- ----- 60CD0510
SWIDB_SB:
61BD629C 0000000084 61BD6220 61BD6318 001 ----- ----- 60CD0510
SWIDB_SB:
61BD6318 0000000028 61BD629C 61BD635C 001 ----- ----- 6087B724
IFINDEX hw
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command, Table 4-10 “show memory Column Description”.

A memory block is considered dead if the process that created the block exits (is no longer around). Each block keeps track of the address and PID of the process that created it. During periodic memory tallying, if the process that the scheduler finds from a block PID does not match the process that the block remembered, the block is marked as dead. Be aware that this memory block may still be in use because it may have been passed to another application before the allocating process was killed. This is typical for configuration processes such as `exec`.

Note Each entry here is for memory that has not been free()'d. The process is gone, but the memory is still allocated. If the memory subsequently gets free()'d, then that entry will not show up in the `show memory dead` output.

The dynamically-calculated sum of the dead memory is also displayed in the **show proc mem** command output in the Holding column, shown here:

PID	TTY	Allocated	Freed	Holding	Getbufs	Retbufs	Process
0	0	14120872	2999080	9184	183356	0	*Dead*

Whenever a process allocates memory, its `totmalloc` count is incremented by the number of bytes allocated; whenever it frees memory, its `totfree` count is incremented. When the process dies, the `totmalloc` count is added to the `dead_totmalloc` count and shows up in the `Allocated` column above. The `totfree` count is added to the `dead_totfree` count and is displayed in the `Freed` column above. The “dead” `Holding` value only applies to memory which is *still allocated* but whose allocating process has died.

For additional information, see:

<http://zed.cisco.com/confluence/display/OSII/IOS+Concepts>

4.7.2.4 show memory allocating-process Output

The **show memory allocating-process** command displays the statistics on the allocated memory with the corresponding allocating process. The **show memory allocating-process** command displays information about memory available after the system image decompresses and loads.

```
.
.
.
Address    Bytes  Prev.      Next     Ref     Alloc Proc    Alloc PC  What
6148EC40   1504  0          6148F24C  1     *Init*       602310FC List
Elements
6148F24C   3004  6148EC40  6148FE34  1     *Init*       60231128 List Headers
6148FE34   9000  6148F24C  61492188  1     *Init*       6023C634 Interrupt
Stack
61492188   44   6148FE34  614921E0  1     *Init*       60C17FD8 *Init*
614921E0   9000  61492188  61494534  1     *Init*       6023C634 Interrupt
Stack
61494534   44   614921E0  6149458C  1     *Init*       60C17FD8 *Init*
6149458C   220  61494534  61494694  1     *Init*       602450F4 *Init*
61494694   4024  6149458C  61495678  1     *Init*       601CBD64 TTY data
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.5 show memory debug references Output

The **show memory debug references** command displays the debug information on references.

```
.
.
.
Address Reference Free_block Cont_block Cont_block_name
442D5774 458CE5EC 458CE5BC 44284960 bss
442D578C 46602998 46602958 44284960 bss
442D58A0 465F9BC4 465F9B94 44284960 bss
442D58B8 4656785C 4656781C 44284960 bss
442D5954 45901E7C 45901E4C 44284960 bss
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.6 show memory ecc Output

The **show memory ecc** command displays the single-bit Error Code Correction (ECC) error logset data. Use this command to determine if the router has experienced single-bit parity errors.

```
.
.
.
Router# show memory ecc
ECC Single Bit error log
-----
Single Bit error detected and corrected at 0x574F3640
- Occured 1 time(s)
- Whether a scrub was attempted at this address: Yes
- Syndrome of the last error at this address: 0xE9
- Error detected on a read-modify-write cycle ? No
- Address region classification: Unknown
- Address media classification : Read/Write Single Bit
  error detected and corrected at 0x56AB3760
- Occured 1 time(s)
- Whether a scrub was attempted at this address: Yes
- Syndrome of the last error at this address: 0x68
- Error detected on a read-modify-write cycle ? No
- Address region classification: Unknown
- Address media classification : Read/Write

Total Single Bit error(s) thus far: 2
```

Table 4-11 describes the significant fields shown in the example.

Table 4-11 show memory ecc Field Description

Column Name	Description
Occured n time(s)	Number of single-bit errors that has occurred.
Whether a scrub was attempted at this address:	Indicates whether a scrub has been performed.
Syndrome of the last error at this address:	Describes the syndrome of last error.
Error detected on a read-modify-write cycle?	Indicates whether an error has occurred.

Table 4-11 show memory ecc Field Description (continued)

Column Name	Description
Address region classification:	Describes the region of the error.
Unknown	
Address media classification:	Describes the media of the error and correction.

4.7.2.7 show memory fast Output

The **show memory fast** command displays the fast memory details for the router. The command displays the statistics for the fast memory. “Fast memory” is another name for “processor memory” and is also known as “cache memory.” Cache memory is called fast memory because the processor can generally access the local cache (traditionally stored on SRAM positioned close to the processor) much more quickly than main memory or RAM.

```

.
.
.
Address      Bytes      Prev      Next Ref      PrevF      NextF Alloc PC what
8404A580 0001493284 00000000 841B6ECC 000 0          84BADF88 815219D8
(coalesced)
841B6ECC 0000020004 8404A580 841BBD18 001 ----- ----- 815DB094
Managed Chunk Queue Elements
841BBD18 0000001504 841B6ECC 841BC320 001 ----- ----- 8159EAC4 List
Elements
841BC320 0000005004 841BBD18 841BD6D4 001 ----- ----- 8159EB04 List
Headers
841BD6D4 0000000048 841BC320 841BD72C 001 ----- ----- 81F2A614 *Init*
841BD72C 0000001504 841BD6D4 841BDD34 001 ----- ----- 815A9514
messages
841BDD34 0000001504 841BD72C 841BE33C 001 ----- ----- 815A9540
Watched messages
841BE33C 0000001504 841BDD34 841BE944 001 ----- ----- 815A95E4
Watched Semaphore
841BE944 0000000504 841BE33C 841BEB64 001 ----- ----- 815A9630
Watched Message Queue
841BEB64 0000001504 841BE944 841BF16C 001 ----- ----- 815A9658
Watcher Message Queue
841BF16C 0000001036 841BEB64 841BF5A0 001 ----- ----- 815A2B24
Process Array
-- More --
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.8 show memory fragment Output

The **show memory fragment** command displays the block details of the fragmented free blocks and allocates the blocks that are physically just before or just after the blocks on the free list.

```

.
.
.
Processor memory
Free memory size : 65566148 Number of free blocks: 230
Address      Bytes      Prev      Next Ref      PrevF      NextF Alloc PC what
645A8148 0000000028 645A80F0 645A8194 001 ----- ----- 60695B20 Init
```

CISCO HIGHLY CONFIDENTIAL

Memory Management Commands

```
645A8194 0000000040 645A8148 645A81EC 000 0 200B4300 606B9614 NameDB
String
645A81EC 0000000260 645A8194 645A8320 001 ----- ----- 607C2D20 Init
200B42B4 0000000028 200B4268 200B4300 001 ----- ----- 62366C80 Init
200B4300 0000000028 200B42B4 200B434C 000 645A8194 6490F7E8 60976574 AAA
Event Data
200B434C 0000002004 200B4300 200B4B50 001 ----- ----- 6267D294 Coproc
Request Structures

6490F79C 0000000028 6490F748 6490F7E8 001 ----- ----- 606DDA04 Parser
Linkage
6490F7E8 0000000028 6490F79C 6490F834 000 200B4300 6491120C 606DD8D8 Init
6490F834 0000006004 6490F7E8 64910FD8 001 ----- ----- 607DF5BC
Process Stack
649111A0 0000000060 64911154 6491120C 001 ----- ----- 606DE82C Parser
Mode
6491120C 0000000028 649111A0 64911258 000 6490F7E8 500770F0 606DD8D8 Init
64911258 0000000200 6491120C 64911350 001 ----- ----- 603F0E38 Init
...

504DCF54 0000001212 504DB2E4 504DD440 001 ----- ----- 60962DFC TCP CB
2C41DCA4 0000000692 2C41BCC8 2C41DF88 001 ----- ----- 60D509BC
Virtual Exec
2C41DF88 0000005344 2C41DCA4 2C41F498 000 504DB2E4 6449A828 60D509BC
(coalesced)
2C41F498 0000000692 2C41DF88 2C41F77C 001 ----- ----- 60D509BC
Virtual Exec
6449A544 0000000692 64499794 6449A828 001 ----- ----- 60D509BC
Virtual Exec
6449A828 0000007760 6449A544 6449C6A8 000 2C41DF88 504D89D4 60D509BC
(coalesced)
6449C6A8 0000008044 6449A828 6449E644 001 ----- ----- 60D2AAC
Virtual Exec
504D8778 0000000556 504D754C 504D89D4 001 ----- ----- 60D4A0B4
Virtual Exec
504D89D4 0000009860 504D8778 504DB088 000 6449A828 504D1B78 60D4A0B4
(coalesced)
504DB088 0000000556 504D89D4 504DB2E4 001 ----- ----- 60D4A0B4
Virtual Exec
504D168C 0000001212 504C9658 504D1B78 001 ----- ----- 60962DFC TCP CB
504D1B78 0000008328 504D168C 504D3C30 000 504D89D4 504C5B54 60962DFC
(coalesced)
504D3C30 0000001212 504D1B78 504D411C 001 ----- ----- 60962DFC TCP CB
504C5870 0000000692 504C5504 504C5B54 001 ----- ----- 60D509BC
Virtual Exec
504C5B54 0000005344 504C5870 504C7064 000 504D1B78 2C423A88 60D509BC
(coalesced)
504C7064 0000000408 504C5B54 504C722C 001 ----- ----- 606E0E44 Chain
Cache No
2C42359C 0000001212 2C41F77C 2C423A88 001 ----- ----- 60962DFC TCP CB
2C423A88 0000008328 2C42359C 2C425B40 000 504C5B54 504D411C 60962DFC
(coalesced)
504E7DD8 0000000828 504E2660 504E8144 001 ----- ----- 60734010
*Packet Header*
65006A08 0000000408 65003834 65006BD0 001 ----- ----- 606E0E44 Chain
Cache No
65006BD0 0000020520 65006A08 6500BC28 000 504E2660 0 60803260
(coalesced)
```

```

6500BC28 0000000828 65006BD0 6500BF94 001 ----- ----- 60734010
*Packet Header*
5C3AE7B8 0000000828 5C3AE614 5C3AEB24 001 ----- ----- 60734010
*Packet Header*
5C3AEB24 0063247532 5C3AE7B8 20000000 000 0 6500C300 60734010
(coalesced)
20000000 0000000828 5C3AEB24 2000036C 001 ----- ----- 60734010
*Packet Header*
6500BF94 0000000828 6500BC28 6500C300 001 ----- ----- 60734010
*Packet Header*
6500C300 0004760912 6500BF94 50000000 000 5C3AEB24 2C42E310 6071253C
(coalesced)
50000000 0000000828 6500C300 5000036C 001 ----- ----- 60734010
*Packet Header*
2C42E0B4 0000000556 2C429430 2C42E310 001 ----- ----- 60D4A0B4
Virtual Exec
2C42E310 0062725312 2C42E0B4 00000000 000 6500C300 0 6071253C
(coalesced)
. .
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.9 show memory multibus Output

The **show memory multibus** command displays the statistics on multibus memory, including the memory-free pool statistics.

```

. .
Address Bytes Prev Next Ref PrevF NextF Alloc PC what
6540BBA0 0000016388 00000000 6540FBD4 001 ----- ----- 60883984 TW
Buckets
6540FBD4 0000016388 6540BBA0 65413C08 001 ----- ----- 60883984 TW
Buckets
65413C08 0000016388 6540FBD4 65417C3C 001 ----- ----- 60883984 TW
Buckets
65417C3C 0000006004 65413C08 654193E0 001 ----- ----- 608A0D4C
Process k
654193E0 0000012004 65417C3C 6541C2F4 001 ----- ----- 608A0D4C
Process k
6541C2F4 0000411712 654193E0 65480B64 000 0 0 608A0D4C
(fragmen)
65480B64 0000020004 6541C2F4 654859B8 001 ----- ----- 608CF99C
Managed s
654859B8 0000010004 65480B64 654880FC 001 ----- ----- 6085C7F8 List
Eles
654880FC 0000005004 654859B8 654894B8 001 ----- ----- 6085C83C List
Heas
654894B8 0000000048 654880FC 65489518 001 ----- ----- 62BF31DC *Init*
. .
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.10 show memory pci Output

The **show memory pci** command displays the statistics on the Peripheral Component Interconnect (PCI) memory.

```

.
.
.
Address      Bytes      Prev      Next Ref      PrevF      NextF Alloc PC what
0E000000 0000000032 00000000 0E000050 000 64F5EBF4 0          00000000
(fragmen)
0E000050 0000000272 0E000000 0E000190 001 ----- ----- 607E2EC0
*Packet *
0E000190 0000000272 0E000050 0E0002D0 001 ----- ----- 607E2EC0
*Packet *
0E0002D0 0000000272 0E000190 0E000410 001 ----- ----- 607E2EC0
*Packet *
0E000410 0000000272 0E0002D0 0E000550 001 ----- ----- 607E2EC0
*Packet *
0E000550 0000000272 0E000410 0E000690 001 ----- ----- 607E2EC0
*Packet *
0E000690 0000000272 0E000550 0E0007D0 001 ----- ----- 607E2EC0
*Packet *
0E0007D0 0000000272 0E000690 0E000910 001 ----- ----- 607E2EC0
*Packet *
0E000910 0000000272 0E0007D0 0E000A50 001 ----- ----- 607E2EC0
*Packet *
0E000A50 0000000272 0E000910 0E000B90 001 ----- ----- 607E2EC0
*Packet *
0E000B90 0000000272 0E000A50 0E000CD0 001 ----- ----- 607E2EC0
*Packet *
.
.
.
Address      Bytes      Prev      Next Ref      PrevF      NextF Alloc PC what
0E000CD0 0000000272 0E000B90 0E000E10 001 ----- ----- 607E2EC0
*Packet *
0E000E10 0000000272 0E000CD0 0E000F50 001 ----- ----- 607E2EC0
*Packet *
.
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.11 show memory processor Output

The **show memory processor** command displays the statistics on the router processor memory.

```

.
.
.
Address      Bytes      Prev      Next Ref      PrevF      NextF Alloc PC what
6540BBA0 0000016388 00000000 6540FBD4 001 ----- ----- 60883984 TW
Buckles
6540FBD4 0000016388 6540BBA0 65413C08 001 ----- ----- 60883984 TW
Buckles
65413C08 0000016388 6540FBD4 65417C3C 001 ----- ----- 60883984 TW
Buckles
65417C3C 0000006004 65413C08 654193E0 001 ----- ----- 608A0D4C
Process k
654193E0 0000012004 65417C3C 6541C2F4 001 ----- ----- 608A0D4C
Process k
6541C2F4 0000411712 654193E0 65480B64 000 0          0          608A0D4C
(fragmen)
65480B64 0000020004 6541C2F4 654859B8 001 ----- ----- 608CF99C
Managed s
```

```

654859B8 0000010004 65480B64 654880FC 001 ----- ----- 6085C7F8 List
Eles
654880FC 0000005004 654859B8 654894B8 001 ----- ----- 6085C83C List
Heas
654894B8 0000000048 654880FC 65489518 001 ----- ----- 62BF31DC *Init*
...

```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.12 show memory scan Output

The **show memory scan** command monitors the number and type of parity (memory) errors on the system.

```

...
Router# show memory scan

Memory scan is on.
Total Parity Errors 1.
Address     BlockPtr     BlckSize   Disposit   Region   Timestamp
6115ABCD   60D5D090    9517A4     Scrubed    Local    16:57:09 UTC Thu Mar 18
...

```

Table 4-12 show memory scan Field Description

Column Name	Description
Address	The byte address where the error occurred.
BlockPtr	The pointer to the block that contains the error.
BlckSize	The size of the memory block.
Disposit	The action taken in response to the error: BlockInUse—An error was detected in a busy block. InFieldPrev—An error was detected in the previous field of a block header. InHeader—An error was detected in a block header. Linked—A block was linked to a bad list. MScrubed—The same address was “scrubbed” more than once, and the block was linked to a bad list. MultiError—Multiple errors have been found in one block. NoBlkHdr—No block header was found. NotYet—An error was found; no action has been taken at this time. Scrubed—An error was “scrubbed.” SplitLinked—A block was split, and only a small portion was linked to a bad list.
Region	The memory region in which the error was found: IBSS—image BSS IData—imagedata IText—imagetext local—heap

Memory Management Commands

Table 4-12 show memory scan Field Description (continued)

Column Name	Description
Timestamp	The time the error occurred.

4.7.2.13 show memory transient Output

The **show memory transient** command displays statistics about transient memory in the system.

```

.
.
.
Address      Bytes      Prev      Next Ref      PrevF      NextF Alloc PC what
81F99C00 0002236408 00000000 821BBC28 000 829C8104 82776FD0 8060B6D0
(coalesc)
821BBC28 0000020004 81F99C00 821C0A7C 001 ----- ----- 8002D5C0
Managed s
821C0A7C 0000010004 821BBC28 821C31C0 001 ----- ----- 811604C0 List
Eles
821C31C0 0000005004 821C0A7C 821C457C 001 ----- ----- 81160500 List
Heas
.
.
.
```

The information displayed for each block in memory is the same as for the **show memory** command; see Table 4-10, “show memory Column Description.”

4.7.2.14 show processes memory Output

The **show processes memory** command displays the sum of allocated memory by process ID.

```

.
.
.
PID TTY Allocated Freed Holding Getbufs Retbufs Process
0 0 12374676 2996700 5913336 0 0 *Init*
0 0 12308 165720 12308 0 0 *Sched*
0 0 844520 3719176 8016 170136 0 *Dead*
1 0 1544 0 8428 0 Chunk Manager
2 0 188 188 3884 0 Load Meter
3 0 340 0 13236 0 Exec
4 0 0 0 6884 0 Check heaps .
.
.
```

The display is in order of the PID.

Table 4-13 displays the **show process memory** column descriptions.

Table 4-13 show process memory Column Description

Column Name	Description
PID	Process ID.
TTY	If associated, the TTY number associated with this PID.
Allocated	Historical total number of bytes allocated by this process, using the <code>malloc()</code> family of functions.
Freed	Historical total number of bytes freed by this process by calling the <code>free()</code> function.
Holding	The total number of bytes currently held by this process.
Getbufs	Historical total number of packet buffers allocated in all issuances of the <code>getbuffer()</code> function.

Table 4-13 show process memory Column Description (continued)

Column Name	Description
Retbufs	Historical total number of packet buffers returned in all issuances of the <code>datagram_done()</code> and <code>retbuffer()</code> functions.
Process	The name of this process.

The difference between Allocated vs. Holding is a common confusion when the Allocated column shows 0 whereas the Holding column shows some amount of memory. Remember that a process need not allocate memory to be the rightful owner. The owner can be changed using the `memory_new_owner()` function.

In fact, this is always done as soon as the process is created—the scheduler allocates the stack, and then bills it to the process that was created. Hence the user can see entries where the allocated column is 0, but there is some amount in the holding of a process.

The command output also contains a line for “Dead” memory. The differences between the **show memory dead** command and this line are discussed below.

PID	TTY	Allocated	Freed	Holding	Getbufs	Retbufs	Process
0	0	844520	3719176	8016	170136	0	*Dead*

The Holding column is dynamically calculated as the sum of all allocated memory for which there is no matching process.

In contrast, the other columns in the **show proc mem** command output are historical totals, collected as the processes are killed. The Allocated column contains the total number of bytes allocated by all processes that have since been killed. The Freed column contains the total number of bytes. This applies as well to the Getbufs and Retbufs columns, which provide historical totals for packet buffer allocation and the packet buffers free for processes that were subsequently killed.

For a more complete description of the **show processes** command and for information on the **show proc epu** command, please see the following URL:

http://www.cisco.com/en/US/docs/ios/fundamentals/command/reference/cf_s2.html

4.7.3 Memory Validation Commands

Use the following hidden command to change the frequency (in seconds) that the `checkheaps` process runs to recalculate the checksum for the text segment in an attempt to catch memory corruption in the text segment:

```
Router(config)# memory validate-checksum nnn
```

This is only useful for:

- images running from memory (not from flash)
- processors that do not provide hardware R/O protection for the image's text segment

Use the following hidden command to change the frequency (in seconds) that the `checkheaps` process runs to validate the headers of every block in dynamically-allocated memory pools:

```
Router(config)# memory check-interval nnn
```

Note that a low value of *nnn* can affect performance.

4.7.4 Malloc-Lite Memory Commands

To expedite `malloc()` and to avoid memory fragmentation for memory blocks of 128 bytes or less, use a feature called “malloc lite.” This feature gives a preallocated chunk when malloc’d for a block of 128 bytes or less.

- For 12.3 and lower:

The feature is turned off by default.

Use the following hidden command to tell the `malloc_lite()` function to use normal `malloc()` infrastructure, instead of the `malloc_lite` infrastructure:

```
Router(config)# memory malloc-lite-use-malloc
```

Use the following hidden command to tell the `malloc()` function to try `malloc_lite()` for small amounts of memory that require an alignment of 4 or less bytes:

```
Router(config)# memory try-malloc-lite
```

- For 12.3T and higher:

The feature is turned on by default.

There is only one new CLI:

```
[no] memory lite
```

Use the following hidden command to tell the `malloc_lite()` function to use normal `malloc()` infrastructure, instead of the `malloc_lite` infrastructure:

```
Router(config)# no memory lite
```

Use the following hidden command to tell the `malloc()` function to try `malloc_lite()` for small amounts of memory that require an alignment of 4 or less bytes:

```
Router(config)# memory lite
```

Note A call to `malloc_lite()` cannot specify an alignment. However, a call to `malloc_aligned()` (which is a variant of `malloc()`) can specify an alignment, and the `malloc_aligned()` function will try `malloc_lite()` only if the alignment requested is 4 or less.

Note The `malloc_named()` function does not use `malloc_lite_internal()` because `malloc_named()` does not work properly with `malloc_lite()`. The name passed by the user to `malloc_named()` is just ignored and a piece of memory is allocated by the `malloc_lite` feature. Refer to the <http://wwwin-metrics.cisco.com/cgi-bin/ddtsdisp.cgi?id=CSCin46246&show=Description> for more information on this. The solution was to bypass `malloc_lite()` in case of `malloc_named()` and call the regular `malloc()` function.

4.7.5 How to Check Blocks and Buffers

show memory summary gives the listing of blocks as seen by the memory manager. The “What” field shows the name of the block or if no name was specified during memory allocation, the memory manager automatically sets the name of the process as the block’s name.

show buffers and its variants gives the listing of the buffers as seen by the buffer pool manager. To list all buffers that are considered to be old, you can have “debug sanity” turned on as soon as the router comes up and then use **show buffers old**.

4.7.6 Garbage Detector

The Garbage Detector (GD) is a tool that can be used to detect leaks in Cisco IOS software. GD is capable of finding leaks in all memory pools, packet buffers, and chunks.

GD works in two modes:

- 1 Normal mode—Where GD uses memory to speed up its operations.
- 2 Low memory mode—Where GD runs without attempting to allocate memory.

The low memory mode is considerably slower than the normal mode and can handle only blocks. That is, there is no support for chunks in low memory mode. The low memory mode is useful when there is no or less memory available on router.

4.7.6.1 Before Using GD

Before using GD, make sure that the following fixes exist in your release:

CSCec88596, CSCed24575, CSCed31959, CSCec58213

Also, make sure you add “memleak-detection” to the attribute field when submitting a DDTs.

4.7.6.2 Using GD

The GD has a simple interface. At any point of time, to get a report of leaks, the GD CLI can be invoked. So, for testing purposes, you may carry out all tests, run all test suites, and at the end of testing, GD can be invoked to get a report on leaks. If you are only interested in the leaks generated by your test cases alone, GD has an incremental option, which can be turned on during the start of testing. After testing completes, you can get a report on only the leaks that occurred after the incremental option was turned on. When submitting bugs based on the reports of GD, add “memleak-detection” to the attribute field of the DDTs.

4.7.6.3 GD Command-Line Interface

The following are the commands relating to GD. They display leaks in different formats:

show memory debug leaks [lowmem | chunk | summary | largest]

set memory debug incremental starting-time [none]

show memory debug incremental {status | leaks [lowmem] | allocations}

4.7.6.3.1 show memory debug leaks

This command invokes normal mode GD. The command displays all blocks that are leaked. However, it doesn't look for leaks in chunks. For example:

```
gt3-7200-1#show memory debug leaks
```

```
Adding blocks for GD...
```

Address	Size	Alloc_pc	PID	Name
				PCI memory

Address	Size	Alloc_pc	PID	Name
				I/O memory

Address	Size	Alloc_pc	PID	Name
				Processor memory

Memory Management Commands

62DABD28	80	60616750	-2	Init
62DABD78	80	606167A0	-2	Init
62DCF240	88	605B7E70	-2	Init
62DCF298	96	605B7E98	-2	Init
62DCF2F8	88	605B7EB4	-2	Init
62DCF350	96	605B7EDC	-2	Init
63336C28	104	60C67D74	-2	Init
63370D58	96	60C656AC	-2	Init
633710A0	304	60C656AC	-2	Init
63B2BF68	96	60C659D4	-2	Init
63BA3FE0	32832	608D2848	104	Audit Process
63BB4020	32832	608D2FD8	104	Audit Process

The output gives the address of the leaked block, the size, the PID of the process that allocated the block, and the `allocator_pc`.

4.7.6.3.2 show memory debug leaks summary

This command summarizes leaks based on `allocator_pc`. The command is similar to **show memory summary**.

For example:

```
gt3-7200-1#show memory debug leaks summary
Adding blocks for GD...

PCI memory

Alloc PC      Size    Blocks     Bytes   What
I/O memory

Alloc PC      Size    Blocks     Bytes   What
Processor memory

Alloc PC      Size    Blocks     Bytes   What
0x605B7E70 0000000032 0000000001 0000000032  Init
0x605B7E98 0000000040 0000000001 0000000040  Init
0x605B7EB4 0000000032 0000000001 0000000032  Init
0x605B7EDC 0000000040 0000000001 0000000040  Init
0x60616750 0000000024 0000000001 0000000024  Init
0x606167A0 0000000024 0000000001 0000000024  Init
0x608D2848 0000032776 0000000001 0000032776  Audit Process
0x608D2FD8 0000032776 0000000001 0000032776  Audit Process
0x60C656AC 0000000040 0000000001 0000000040  Init
0x60C656AC 0000000248 0000000001 0000000248  Init
0x60C659D4 0000000040 0000000001 0000000040  Init
0x60C67D74 0000000048 0000000001 0000000048  Init
```

The output here is summarized based on `allocator_pc` and then on the size of the block.

The output doesn't list out blocks that are leaked, but rather just summarizes the `allocator_pcs` that are leaking, the total size of the leak, the number of blocks leaked, and the size of the blocks. The size of the block is under the `Size` column, the number of blocks leaked is under `Blocks`, and the total amount of memory leaked is under `Bytes`.

4.7.6.3.3 show memory debug leaks chunks

This command invokes normal mode GD, looking for leaks in chunks as well. The report is similar to the output from **show memory debug leaks**, except that it includes a report on chunk elements that are leaked as well. For example:

```
gt3-7200-1#show memory debug leaks chunks
Adding blocks for GD...

PCI memory
Address      Size     Alloc_pc   PID   Name

Chunk Elements:
Address      Size     Parent     Name

I/O memory
Address      Size     Alloc_pc   PID   Name

Chunk Elements:
Address      Size     Parent     Name

Processor memory
Address      Size     Alloc_pc   PID   Name
62DABD28      80    60616750  -2   Init
62DABD78      80    606167A0  -2   Init
62DCF240      88    605B7E70  -2   Init
62DCF298      96    605B7E98  -2   Init
62DCF2F8      88    605B7EB4  -2   Init
62DCF350      96    605B7EDC  -2   Init
63336C28      104   60C67D74  -2   Init
63370D58      96    60C656AC  -2   Init
633710A0      304   60C656AC  -2   Init
63B2BF68      96    60C659D4  -2   Init
63BA3FE0      32832 608D2848  104  Audit Process
63BB4020      32832 608D2FD8  104  Audit Process

Chunk Elements:
Address      Size     Parent     Name
62D80DA8      16    62D7BFD0 (Managed Chunk )
62D80DB8      16    62D7BFD0 (Managed Chunk )
62D80DC8      16    62D7BFD0 (Managed Chunk )
62D80DD8      16    62D7BFD0 (Managed Chunk )
62D80DE8      16    62D7BFD0 (Managed Chunk )
62E8FD60      216   62E8F888 (IPC Message He)
```

The chunk leak report contains the name of the chunk, the address of the leaked element, its size, and the parent chunk to which it belongs.

The following steps show an example of how to debug a packet element leak.

Snip of **show memory debug leaks chunks** output<leaked chunk elements>

```
Address      Size     Parent     Name
648C052C      8    648BE938 (Packet Element)
648C0538      8    648BE938 (Packet Element)
648C0568      8    648BE938 (Packet Element)
.....
648C0874      8    648BE938 (Packet Element)
648C0898      8    648BE938 (Packet Element)
648C08EC      8    648BE938 (Packet Element)
```

Step 1 Find for whom the packet element is given.

Dump a leaked chunk element:

```
ios104#sh mem 0x648C06F4 0x648C06FC  
648C06F0: 648C06C4 5063CD8C 64 d..DPcM.d
```

Step 2 Find the start of the block where the “pointer to data” belongs to.

Each element chunk (of type `elementtype_`) has two fields “next pointer” and “pointer to data”.

Here 5063CD8C is the pointer to data.

```
ios104#sh mem 0x5063CC00 | i AB1234  
5063CC00: 00000000 FD0110DF AB1234CD FFFE0000 ....}...+..4M.~..  
5063D330: AB1234CD FFFE0000 00000000 630416A8 +.4M.~.....c..(.  
5063D990: 635691B4 FD0110DF AB1234CD FFFE0000 cV.4}...+..4M.~..
```

The leaked element points to the block starting at 0x5063CC08.

Step 3 Dump the block and decode the allocator_pc.

```
ios104#sh mem 0x5063CC00  
5063CC00: 00000000 FD0110DF AB1234CD FFFE0000 ....}...+..4M.~..  
5063CC10: 00000000 630416A8 619CDDF4 5063D330 ....c..(a.)tPcS0  
5063CC20: 5063CB0 8000037C 00000001 00000000 PCKP...|.....  
5063CC30: 00000001 648280D8 0000004A 00000000 ....d..X...J....  
5063CC40: 00000000 00000000 00000000 63E9371C .....ci7.  
5063CC50: 6356918C 648C08E0 648C08B0 00000003 cV..d..`d..0....
```

Decoding the allocator_pc

```
Enter hex value: 619CDDF4  
0x619CDDF4:xml_db_tag_array_create(0x619cddb4)+0x40
```

Step 4 Dump other leaked chunk elements (if any) and find the user of the packet element.

```
ios104#sh mem 0x648C0568 0x648C056F  
648C0560: 648C0538 5063CE34 d..8PcN4  
ios104#sh mem 0x648C0898 0x648C089F  
648C0890: 648C0874 5063CC54 d..tPcLTios104#sh mem  
0x648C07F0 0x648C07FF  
648C07F0: 648C07D8 5063CCCC 648BE938 00000000 d..XPcLLd.i8....
```

Note Sometimes the pointer to data is NULL. If so, check the other leaked chunk elements. At least one leaked chunk element will have a pointer to a block

Sometimes chunk leaks show Alloc PC as NA. You may wonder how to debug such leaks.

```
router2#sh mem deb le ch  
Adding blocks for GD...
```

Processor memory

Address	Size	Alloc_pc	PID	Alloc-Proc	Name
---------	------	----------	-----	------------	------

Chunk Elements:

AllocPC	Address	Size	Parent	Name
NA	8E5E7F8	8	8E52B08	(IPSM Octet Str)
NA	8E5E800	8	8E52B08	(IPSM Octet Str)
NA	8E5E828	8	8E52B08	(IPSM Octet Str)
NA	8E5E830	8	8E52B08	(IPSM Octet Str)
NA	8E5E858	8	8E52B08	(IPSM Octet Str)
NA	8E5E860	8	8E52B08	(IPSM Octet Str)
NA	8E5E888	8	8E52B08	(IPSM Octet Str)
NA	8E5E890	8	8E52B08	(IPSM Octet Str)
NA	8E5E8B8	8	8E52B08	(IPSM Octet Str)
NA	8E5E8C0	8	8E52B08	(IPSM Octet Str)

If the chunk element being leaked is of type non-big header chunk, then NA is printed for allocator PC. Cisco IOS stores the allocator PC of a chunk element only for big header chunks.

4.7.6.3.4 show memory debug leaks largest

This command displays the top ten leaking allocator_pcs and the total amount of memory they have leaked. Additionally, each time it is invoked it remembers the previous invocation's report and compares the report of the previous invocation against the current invocation's report. If there are new entries in the current report they are tagged as inconclusive. If the same entry appears in the previous invocation's report as well as in the current invocation's report, the inconclusive tag is not added. As suggested earlier, it would be beneficial to run GD more than once and take only the consistently reported leaks. Running GD would display the top ten leaking allocator_pcs and would also display inconsistent entries as inconclusive.

For example:

```
gt3-7200-1#show memory debug leaks largest
Adding blocks for GD...
```

Alloc_pc	PCI memory	total leak size
Alloc_pc	I/O memory	total leak size
Alloc_pc	Processor memory	total leak size

Alloc_pc	Processor memory	total leak size
608D2848	32776	inconclusive
608D2FD8	32776	inconclusive
60C656AC	288	inconclusive
60C67D74	48	inconclusive
605B7E98	40	inconclusive
605B7EDC	40	inconclusive
60C659D4	40	inconclusive
605B7E70	32	inconclusive
605B7EB4	32	inconclusive
60616750	24	inconclusive

```
gt3-7200-1#show memory debug leaks largest
Adding blocks for GD...
```

Alloc_pc	PCI memory	total leak size
Alloc_pc	I/O memory	

```

Alloc_pc      total leak size

          Processor memory
Alloc_pc      total leak size
608D2848    32776
608D2FD8    32776
60C656AC    288
60C67D74    48
605B7E98    40
605B7EDC    40
60C659D4    40
605B7E70    32
605B7EB4    32
60616750    24

```

4.7.6.3.5 show memory debug leaks lowmem

This command forces GD to work in low memory mode. The amount of time taken for analysis is considerably greater than that of normal mode. The output is similar to **show memory debug leaks**. You can use this command when you already know that normal mode fails (perhaps by an unsuccessful previous attempt to invoke normal mode GD).

4.7.6.3.6 set memory debug incremental starting-time

For incremental analysis, you need to define a starting point. **set memory debug incremental starting-time** sets the current time as the starting time of analysis.

Once set, only memory allocated after the issue of **set memory debug incremental starting-time** is reported as a leak.

4.7.6.3.7 show memory debug incremental status

This command displays whether the starting point of incremental analysis has been defined and the time elapsed since then. For example:

```

gt3-7200-1#show memory debug incremental status
Incremental debugging is enabled
Time elapsed since start of incremental debugging: 00:00:10

```

4.7.6.3.8 show memory debug incremental leaks

This command is the same as **show memory debug leaks** except that it only displays memory that was leaked after the issue of **set memory debug incremental starting-time**.

4.7.6.3.9 show memory debug incremental leaks lowmem

This command forces GD to work in low memory mode. The output is similar to **show memory debug leaks lowmem** except that it only displays memory that was leaked after the issue of **set memory debug incremental starting-time**.

4.7.6.3.10 show memory debug incremental allocation

This command displays all memory blocks that were allocated after the issue of **set memory debug incremental starting-time**. These are just memory allocations. They are not necessarily leaks. For example:

```
gt3-7200-1#show memory debug incremental allocations
Address      Size Alloc_pc PID Name
62DA4E98     176 608CDC7C 44   CDP Protocol
62DA4F48     88 608CCCC8 44   CDP Protocol
62DA4FA0     88 606224A0 3    Exec
62DA4FF8     96 606224A0 3    Exec
635BF040     96 606224A0 3    Exec
63905E50    200 606A4DA4 69   Process Events
```

All commands relating to GD invoke normal mode GD, except when the low memory option is specifically mentioned. In normal mode, if GD discovers that there is insufficient memory to proceed in normal mode, it displays an appropriate message and switches to low memory mode.

4.7.6.4 False Alarms

There are a few cases where GD can report false alarms. In general the cases are rare, except for race conditions that cause memory to be incorrectly reported as leaks.

The following is a list of cases where false alarms can occur:

- 1 When memory is allocated and the pointer is prevented from pointing to the block. For example:

```
ptr = malloc(100);
ptr--;
```

You can always do `ptr++` and then free the block. However, to GD, the block allocated would seem to be leaked. However, such programming constructs are not common.

- 2 When memory is allocated and the pointer is stored in a location that is not under the control of the IOS memory manager.

For any questions, suggestions, and support for GD, send an email to memleak-detection@cisco.com. Subscription to the alias is open to all and future developments on GD are announced on this alias.

4.8 Dynamic Bitfield Management

Dynamic bitfield management provides support for whenever a bit is to be set in a bitfield and the bitfield is not wide enough (or does not exist). A new bitfield is allocated and the old bits are copied into the new bitfield. Bitfield expansion can only occur at the process level. The dynamic bitfield management API allows for the setting, clearing, and testing of bits. Nonexistent bits are implicitly zero, so a test always returns `FALSE` if the bit position to be tested is beyond the current bitfield size.

Manipulation of variable-length bitfields is supported. The allocation and resizing of the bitfields is automatic and hidden from the caller. The external data structure is the `dynamic_bitfield` structure. All operations in this module are performed on this structure. Callers should embed a `dynamic_bitfield` structure wherever a variable-length bitfield is needed, and then simply call the manipulation routines (see the following “List of Bitlogic API Functions”). The caller *must* call `bitfield_destroy()` when the caller is done using a bitfield in order to return the allocated memory; otherwise, memory leaks will result.

Bitfields are allocated out of dynamic chunks to minimize memory wastage (since 256 interfaces require only 32 bytes of bitfield, and the `malloc` overhead is ~40 bytes). This creates some difficulty in increasing the size because chunks are by definition fixed in size. To get around the fixed size, a new chunk pool of larger size chunks is dynamically created when needed. The pools are chained

together with the internal `bitfield_pool_link()` call. A queue of `bitfield_pool` structures is used to maintain these pools. The pools on the `bitfield_pools` queue are sorted in inverse order by size (the first one on the queue is the largest).

4.8.1 List of Bitlogic API Functions

The Cisco IOS API functions provided for dynamic bitfield management are as follows:

- `bitfield_check()`
- `bitfield_clear()`
- `bitfield_clear_many()`
- `bitfield_clearmask()`
- `bitfield_destroy()`
- `bitfield_find_first_clear()`
- `bitfield_find_first_set()`
- `bitfield_lock()`
- `bitfield_set()`
- `bitfield_set_many()`
- `bitfield_setmask()`
- `bitmask_any_bits_in_first_are_set_in_second()` (*New in 12.3T*)

Note Dynamic bitfields cannot be expanded at interrupt level. However, these bitfield API functions (with the exception of `bitfield_destroy()`) can be called from an interrupt if they do not cause the bitfield to expand.

4.8.2 Checking Bitfields

To test if a bit in the bitfield is set, call the `bitfield_check()` function. This function tests a bit in a bitfield and returns the bit value or returns FALSE if the bitfield is too small.

```
#include "bitlogic.h"
boolean bitfield_check(ulong bitnum, dynamic_bitfield *bitfield);
```

4.8.3 Clearing Bitfields

To clear a bit in a bitfield, call the `bitfield_clear()` function.

```
#include "bitlogic.h"
void bitfield_clear(ulong bitnum, dynamic_bitfield *bitfield);
```

To clear many bits in a bitfield, call the `bitfield_clear_many()` function.

```
#include "bitlogic.h"
void bitfield_clear_many(dynamic_bitfield *bitfield, ulong count, ...);
```

To find the first clear bit in a bitfield, call the `bitfield_find_first_clear()` function.

```
#include "bitlogic.h"
boolean bitfield_find_first_clear(dynamic_bitfield *bitfield,
                                  ulong *bitnum);
```

To clear bits in a bitfield using a supplied mask, call the `bitfield_clearmask()` function.

```
#include "bitlogic.h"
void bitfield_clearmask(dynamic_bitfield *mask, dynamic_bitfield *bitfield);
```

4.8.4 Destroying Bitfields

To destroy a bitfield, call the `bitfield_destroy()` function.

```
#include "bitlogic.h"
void bitfield_destroy(dynamic_bitfield *bitfield);
```

4.8.5 Setting Bitfields

To set a bit in a bitfield, call the `bitfield_set()` function.

```
#include "bitlogic.h"
void bitfield_set(ulong bitnum, dynamic_bitfield *bitfield);
```

To find the first set bit in a bitfield, call the `bitfield_find_first_set()` function.

```
#include "bitlogic.h"
boolean bitfield_find_first_set(dynamic_bitfield *bitfield, ulong *bitnum);
```

To set many bits in a bitfield, call the `bitfield_set_many()` function.

```
#include "bitlogic.h"
void bitfield_set_many(dynamic_bitfield *bitfield, ulong count, ...);
```

To set bits in a bitfield using a supplied mask, call the `bitfield_setmask()` function.

```
#include "bitlogic.h"
void bitfield_setmask(dynamic_bitfield *mask, dynamic_bitfield *bitfield);
```

Note Bitmasks should be put at the end of structures unless you are sure of their exact length.

To determine if any of the bits that are set in one bitfield are also set in another bitfield, call the `bitmask_any_bits_in_first_are_set_in_second()` function. (*New in 12.3T*)

```
#include "bitlogic.h"
extern boolean
bitmask_any_bits_in_first_are_set_in_second(dynamic_bitfield *a,
                                            dynamic_bitfield *b);
```

4.8.6 Locking Bitfields

To lock a bitfield, call the `bitfield_lock()` function.

```
#include "bitlogic.h"
boolean bitfield_lock(dynamic_bitfield *bitfield);
```

4.9 Dynamic Bitlists

The bitlist API is used to create and maintain dynamic bitlists. Dynamic bitlists can be used to develop platform-specific loopback detection code (for example, see ENG-75974).

4.9.1 The Bitlist API

The number of bits parameter and the base bit parameter are passed to several of the bitlist API functions. These two parameters are described here:

- 1 The number of bits parameter is the number of different bits in the bitlist.
- 2 All bits in a bitlist are numbered. The base bit parameter is passed to the API functions that initialize the bitlist data structure. The base bit parameter specifies what is the bit number of the lowest-numbered bit in the list. In general, bit numbers in the bitlist start at <base_bit> and extend to <base_bit + number_of_bits - 1>. For example, if *base_bit* is 10 and *number_of_bits* is 100, the bits in the bitlist are numbered from 10...109.

The base bit should always be ≥ 0 , because some of the bitlist API functions use a return value of -1 to indicate that a bit was not found.

The bitlist API routines are not intended to be used directly in most cases, but rather through a set of wrapper functions.

Note Although the API was intended to be used with wrapper functions, there are some significant pieces of code that use the bare API without wrappers. This is because the author of the code wanted to be able to use bitlists of varying sizes and varying base bit numbers without chewing up a lot of memory by simply making all the bitlists big enough to hold every possible bit number.

The API reference page includes detailed information on each of the following bitlist API functions:

- 1 `bitlist_alloc()`
- 2 `bitlist_and()`
- 3 `bitlist_and_not()`
- 4 `bitlist_clear()`
- 5 `bitlist_clearall()`
- 6 `bitlist_copy()`
- 7 `bitlist_count()`
- 8 `bitlist_equal()`
- 9 `bitlist_find_after()`
- 10 `bitlist_find_bit()`
- 11 `bitlist_find_first()`
- 12 `bitlist_find_next()`
- 13 `bitlist_free()`
- 14 `bitlist_get()` (It is *not* recommended to use this function, use `bitlist_test()` instead.)
- 15 `bitlist_or()`
- 16 `bitlist_set()`
- 17 `bitlist_setall()`

```

18 bitlist_set_fromrange()
19 bitlist_test()
20 bitlist_validbit()
21 bitlist_xor()

```

4.9.2 Bitlist Wrapper Functions

The wrapper functions should enforce type-checking and do some bookkeeping so the user does not have to.

The following example will create a set of wrappers for a bitlist with 128 bits that is named “xyz_bitlist_t”:

Note

If XYZ_BASE is set to 0 then the possible bit values are 0 .. 127.

If XYZ_BASE is set to 1 then the possible bit values are 1 .. 128.

If XYZ_BASE is set to n then the possible bit values are n .. (n + 128 - 1).

```

#define XYZ_BASE          0
#define XYZ_MAX_BITS     128
DEF_BITLIST(xyz_bitlist_t, XYZ_MAX_BITS);

static inline boolean
xyz_bitlist_test(xyz_bitlist_t *bl, int bit)
{
    return bitlist_test((bitlist_t *) bl, bit);
}

static inline void
xyz_bitlist_set(xyz_bitlist_t *bl, int bit)
{
    bitlist_set((bitlist_t *) bl, bit);
}

static inline void
xyz_bitlist_clear(xyz_bitlist_t *bl, int bit)
{
    bitlist_clear((bitlist_t *) bl, bit);
}

static inline void
xyz_bitlist_setall(xyz_bitlist_t *bl)
{
    bitlist_setall((bitlist_t *) bl, XYZ_MAX_BITS, XYZ_BASE);
}

static inline void
xyz_bitlist_clearall(xyz_bitlist_t *bl)
{
    bitlist_clearall((bitlist_t *) bl, XYZ_MAX_BITS, XYZ_BASE);
}

static inline int

```

```

xyz_bitlist_size(void)
{
    return XYZ_MAX_BITS;
}

static inline int
xyz_bitlist_find_first(xyz_bitlist_t *bl)
{
    return bitlist_find_first((bitlist_t *) bl);
}

static inline int
xyz_bitlist_find_next(xyz_bitlist_t *bl)
{
    return bitlist_find_next((bitlist_t *) bl);
** }

static inline int
xyz_bitlist_find_after(xyz_bitlist_t *bl, int bit)
{
    return bitlist_find_after((bitlist_t *) bl, bit);
}

static inline void
xyz_bitlist_copy(xyz_bitlist_t *from, xyz_bitlist_t *to)
{
    bitlist_copy((bitlist_t *) from, (bitlist_t *) to);
}

```

Note `xyz_bitlist_find_bit` is not included here because the `bitlist_bit()` function is used in the `bitlist_find_after()`, `bitlist_find_first()`, and `bitlist_find_next()` functions.

`xyz_bitlist_validbit()` is not included here because the `bitlist_validbit()` function is used in the `bitlist_test()`, `bitlist_set()`, and `bitlist_clear()` functions.

Also, `xyz_bitlist_get` is not included here because it is recommended to use `xyz_bitlist_test` instead.

4.9.3 Performance Analysis of Bitlist Algorithms

For information on the performance analysis results for several algorithms that implement the `bitlist_find_bit()` functionality, see EDCS-347779.

However, in any given IOS branch, `bitlist_find_bit()` will be coded to use only one algorithm; there is no way for the user of the bitlist code to pick and choose among several algorithms. At the time of this writing (December 2005), the latest hawaii code is using an old, non-optimized algorithm that is even less efficient than the “original” algorithm listed in EDCS-34779 for `bitlist_find_bit()`.

4.9.4 Safe Bitlist API Functions

There are multiple bitlist API routines that write state into a bitlist header. In addition to “housekeeping” information, such as the number of elements or bits and the base bit, two additional fields, `last_elem` and `last_bit`, have traditionally been used to store information from a prior bitlist search, or to pass information to a subsequent search—for example, when calling the `bitlist_find_next()` function. This information is generally helpful when iterating through all the set bits in a bitlist in a for or while loop to perform some operation on each related entity.

However, a problem can occur because although the bitlist state of `last_elem` and `last_bit` is global, it really only has local significance. If a process is suspended during the processing of a bitlist, another process can operate on the same bitlist and update the global state of the bitlist represented by `last_elem` and `last_bit`. When the original process is rescheduled, it uses the global state that has been changed by unrelated code and processing is resumed on the bitlist at the wrong location. Bits can be processed more than once or can be skipped entirely. Suspending a process during bitlist traversal is quite common. For example, the `bitlist_to_string()` function is usually called from within a loop and the work done on a set bit often involves calling `printf`, resulting in a process suspension.

An additional problem with these bitlist routines can occur in IONized code when a bitlist is shared between different processes using shared memory. Code with read-only access to a shared memory segment cannot call routines like `bitlist_find_next()` because the state cannot be written to the bitlist. As in the IOS process suspension example, reading state information from `last_elem` and `last_bit` does not make sense for an ION process when the information can be written by different processes.

As a result of these potential problems, the following “safe” functions are recommended:

- `bitlist_safe_find_first` (*currently in 12.2SX only*)
- `bitlist_safe_find_after` (*currently in 12.2SX only*)
- `bitlist_safe_to_string` (*currently in 12.2SX only*)

These safe functions rely on the local context being passed to infer state instead of using a global state. They also enable bitlists to be used in shared memory.

Note Since the use of the bitlist APIs are quite widespread in the codebase, an attempt to change all of them to safer alternatives is incredibly challenging and somewhat risky. However, it is presumed that sometime in the future the `last_elem` and `last_bit` fields can be eventually be deleted after the safe bitlist functions are predominately in place.

4.10 Virtual Memory

As of Release 12.0, Cisco IOS software provides support for all MIPS and other platforms with virtual memory (VM). However, note that IOS has nonvirtual memory and does not do demand paging. Instead, IOS only does a sort of demand page-in of readonly pages. 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 the initial writing (September 1998), VM was in its first “incarnation.” A project was underway to reduce the porting and maintenance effort required by the current .link file and image creation methods; however, the project has not been completed. The status of VM was revisited in May 2003 and VM had not changed significantly in Release 12.1, 12.2, or 12.3; therefore, this documentation remains valid.

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.

4.10.1 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.

4.10.1.1 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.

4.10.1.2 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.

4.10.2 Overview of Cisco IOS VM

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

4.10.2.1 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.

4.10.2.2 Benefits and Costs

Adding VM to your platform yields two benefits: it decreases the minimum DRAM required to run the Cisco 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 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 at the expense of performance and, perhaps, live Cisco IOS flash burn. A typical development effort takes 2 to 3 months.

4.10.3 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.

4.10.4 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 data 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?”

4.10.5 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.

4.10.5.1 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. Both the TEXT and LOCAL areas are virtual for Cisco platforms that support VM and the VM is used primarily to page out and page in pages in the TEXT segment to save space.

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.

4.10.5.2 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.

4.10.5.3 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.

4.10.6 Porting VM to a Platform

Step 1 Make it link and run using the various stub headers in `vm_port.h` (e.g., `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 2 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 3 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 4 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 5** 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 6** 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 7** 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 8** 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.

4.10.7 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.

4.10.8 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 4-14 provides a brief description:

Table 4-14 Mathematical Notations in VM Code

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

4.10.9 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://foldoc.org>). 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 memory*. 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 prepping (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.

4.11 Caching Issues

There are CPU caches and packet caches. Sometimes routers have a high speed special packet store which can also be called cache, and this is very different from dedicated CPU cache memory. The CPU cache memory is not really accessible as memory, but is used to hide memory latencies, and is often automatically managed by the CPU.

Here are the different types of memory access:

- 1 Cached Read, where a read request from the CPU to a memory address will go through the cache. If the cache contains a copy of that memory location, it is returned immediately. If it doesn't, a read to the memory location will be issued and the results will be placed in the cache.
- 2 Uncached Read, where a read request from the CPU bypasses the cache entirely.
- 3 Write with cache write-back, where a write request from the CPU will store the value in the cache (if the memory location doesn't appear in the cache, it will be fetched first), but the memory location is not updated.
- 4 Write with cache write-through, where a write request from the CPU will store the value in the cache *and* also issue a write operation to the memory location.

Note It is not quite correct to say that the write-back to cache does not propagate to RAM, since eventually the data will likely get back to RAM, it is just that it may not happen during that write-to-cache operation. The write-through to cache will not only update the data in the cache, but will also generate a memory transaction to update the data in RAM.

- 5 Uncached Write, where a write request will bypass the cache entirely, and write directly to RAM.

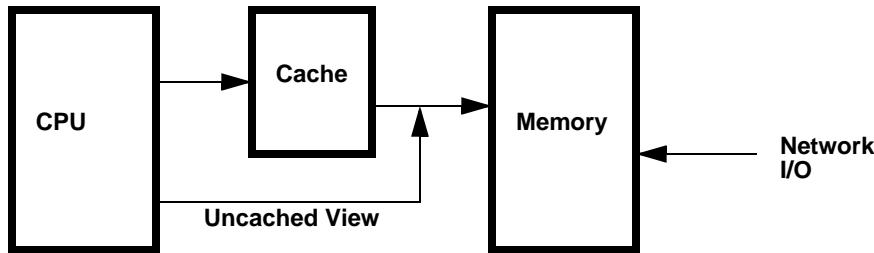
There are also other cache operations that CPUs can explicitly issue, such as:

- 1 Cache invalidate, where a cache entry can be forced to be marked as invalid so that the contents are lost; note that this does not flush the contents of the cache entries back to memory.
- 2 Cache prefetch, where the CPU can request a memory location to be read into cache speculatively in the expectation that at some time later the CPU will require this memory location. Not all CPUs support cache prefetch, but with data such as packet data, cache prefetching is a very powerful technique for improving performance, in effect reducing or eliminating CPU stalls due to memory latency.

4.11.1 Invalidated Cache

The invalidated cache does not write back to RAM, but simply invalidates the cache lines. This is quite important, and to illustrate this, consider Figure 4-8:

Figure 4-8 IOS Caching Diagram



Imagine the following typical situation:

- 1 Network I/O dumps a packet to memory, so the CPU gets an interrupt.
- 2 The CPU can read/modify/write packet data via the uncached view of memory, but this is slow, since every access requires a memory transaction.
- 3 The CPU uses cached-write-through view of packet memory, meaning that the packet data can be accessed via the cache, making it much faster.
- 4 When new packet data arrives, however, the cache is out of sync with the memory, so the CPU has to explicitly invalidate the cache so that the old cache data disappears and the new memory data can be read in. This is not a write-back but a true invalidation, and the old data that was in the cache is lost. When a cache entry is set up as write-through, no write-back is done.

The details of how the various cache operations interact with memory can be CPU- and architecture-specific. Even in the CPU family it can vary, depending on what the architecture is. For example, the BCM-1250 (used on the NPE-G1) is a MIPS CPU, but since it is a completely “cache coherent” memory system, no cache invalidations are required (“cache coherent” means every memory transaction will also automatically tell the CPU caches that new memory data is available, and so any cache lines for that memory location can be invalidated).

For the 7500, packets are not copied into RP memory unless they need to be because MEMD is running short. Sometimes when packets are queued for transmit, they are copied into the “backing store”, which is essentially RP memory. Note that no packet can be transmitted unless the packet is in MEMD, so packets copied into RP memory must be copied back into MEMD buffers before the packets can be transmitted.

Obviously, since the overhead is high on copying the packet into cache, it does not make sense for each packet received to be copied in. However, for NBAR (Network Based Application Recognition, which refers to the classification engine that performs the stateful inspection on packets), we “invalidate” the cache before looking at each packet because NBAR performs deep inspection in the packet and needs to make sure that the Application-layer information is valid.

The RP memory will mark the cache as “invalid” so that the next access will read the data from RAM into cache; that should insure the latest (and valid) copy. This is normally what happens. However, it gets expensive if every feature has to do an invalidation before accessing packet data, since that defeats a lot of the benefits of having a cache in the first place. When the cache gets refreshed, the complete packet in the cache does not necessarily need to be refreshed. Instead, you can specify the number of bytes to refresh.

If you specify a particular number of bytes with the 7200, it will round down the number to an even cache line (32 bytes) and then make sure that at least that number of bytes are cleared. If (start, bytes) is 0x0000005, 5, it will clear 0x0000000 for 32 bytes (a cache line). If (start, bytes) is 0x0000001f, 5, it will clear 0x0000000 for 64 bytes; that is making sure that at least the 5 bytes that you request are cleared. It is worthwhile to spend time on this because it can be a problem.

So, an issue for any platform that uses a cached view of packets, like any platform that uses a MIPS or a PPC, is how the application knows whether it needs to invalidate the cache and refresh.

4.11.2 History of IOS Caching

There is a definite IOS caching issue that will become more and more obvious as time goes on. Here is some IOS caching history that is provided to help explain how the issue came about:

- 1 Traditionally, all Cisco IOS code used uncached views of packet memory because CPU performance and memory performance were relatively equal (in the 680x0 timeframe). Also, packet memory was small and sometimes in higher speed SRAM.
- 2 When CPUs like the MIPS were starting to be extensively used (starting with the 4500 and 7500), testing showed how much faster packets could be switched if the packet headers were accessed via a cached view. There were a couple of reasons for this. One reason was that more and more features required a relatively large number of accesses to the packet header (for example, ACLs, NAT, Netflow, etc.), and the second reason was that CPU speeds started to far outstrip memory performance, so that the instruction cycle cost of a memory access operation started to grow.

However, the existing IOS switching path *still* treated everything as being in an uncached view of the packet header.

- 3 The platform developers solved this by creating separate “optimum” and “fast” switching paths, where the driver and platform-specific front end to the IOS switching path itself modified the packet header pointer to access the packet via a cached view of the packet, without the IOS code knowing about it.
- 4 There were some caveats. Because of the invalidation problem mentioned above, the driver needed to invalidate all the cache lines that were touched by the switching path, lest there be stale data in them. Since caching was only used on the “optimum” IP path, there was a guarantee that only 3 cache lines were ever touched. If the packet was not a simple IP packet to be switched, the normal uncached view was used and the packet fast switched as normal.

As long as the platform driver existed at the top and tail of the switching path, this worked fine. The driver could invalidate 3 cache lines, and the IOS switching path never had to know or understand about cached or uncached views, and the switching performance was very good.

- 5 Because this was all done outside the switching path, on the platform edges, so to speak, rather than as an architectural change to the IOS switching code, these cached/uncached views are embodied in drivers throughout platform code.
- 6 When the CEF code was integrated, replacing the “optimum” path, the 3-cache-line assumption was maintained.
- 7 When CPUs were available that allowed cache-line prefetching, prefetching was used to make switching even faster. Because of the significant performance gains obtained by using a cached view of the packet with prefetching, there is little chance of reverting to code that exclusively uses an uncached view of the packet.

4.11.3 Current IOS Caching Issues

Now we have a situation where more and more features or protocols need to look deeper inside the packet header, going past the 3 cache lines of invalidated data, and these features are designed to run in the CEF path since that is the preferred path for new features. So features like NBAR handle cache invalidations in a special way.

Some of the current issues are:

- How to move the cached/uncached view infrastructure out of the drivers (where it certainly does not belong).
- How to move the IOS switching path code along and start to expose some of these caching advantages to the switching path.

It is very likely that in the future, the feature writers will need to be aware of these caching issues, and an IOS infrastructure will need to be developed to provide a CPU- and platform-independent API to these caching facilities.

4.12 ID Manager

The ID Manager allows client applications to create tables of ID values from which clients can request an ID to be allocated and associated with a memory pointer. Upon reception of an ID, the ID Manager is used to convert the ID to the memory pointer. The ID Manager resolves the problem known as “freeing twice” (when the coder is not aware or is negligent of some process or processes “waiting” for the same memory location and her/his code frees it first before the process or visa versa, causing a crash) by making memory pointers unavailable except via the ID Manager. The ID Manager is available in 12.2S, 12.3, 12.3T, and 12.4 (the most enhanced version is in 12.4T).

As a client of the ID Manager, your application is able to define its own tables and can have any number of ID tables up to the system maximum. The system maximum number of ID tables is 512 (increased from 128 to 512 in 12.4T). The maximum number of IDs per table is dependant on the dimensions of the ID table that is specified at the time of the creation of the ID table.

The typical scenario for using an ID table is as follows:

- 1 `client_A` creates an ID table.
- 2 `client_A` associates a pointer to memory with an ID by asking the ID Manager to allocate an ID for the pointer in `client_A`'s ID table.
- 3 `client_A` gives out this ID to other clients (for example, `client_B`) instead of the pointer.
- 4 `client_B` is prevented from directly accessing the memory that is associated with this ID, thereby preventing unsafe operations, such as freeing the memory that `client_A` still holds the reference to.
- 5 If `client_B` wants any details about the ID, `client_B` will have to provide the ID to `client_A` and request the specific information. In this case, the ID is more like a handle (for information on the handle subsystem, see EDCS-306135).

As a result of your application code's capability to use an ID with the `id_to_ptr()` function, instead of the pointer to the actual memory location directly, the code is more robust because it eliminates the security-related vulnerability in networking devices caused by “freeing twice”. Another characteristic of handles is that they are opaque, so `client_B` cannot dereference or otherwise become dependent on the structures used by `client_A` when storing `client_A`'s state when the ID Manager is used.

An ID from the ID Manager consists of three parts, as shown in Figure 4-9.

Figure 4-9 Parts of an ID from the ID Manager

tableSequence	tableNumber	tableIndex
---------------	-------------	------------

During the creation of the ID table, the ranges of values for `tableNumber` (`numTableSlot`) and `tableIndex` (`numIDsPerSlot`) are specified. But, `tableSequence` is automatically generated and by default `tableSequence` is randomized. In case the client wants a sequence of IDs, then the client can specify `id_table_no_randomize_id()` (which will keep the `tableSequence` part always at zero).

Randomization of `tableSequence` makes the IDs vary anywhere in the 32 bit data range (1 to $2^{32}-1$). If the client wants to have the IDs allocated sequentially, randomization of the `tableSequence` should be turned off. Having the `tableSequence` turned off has a side effect, which is that the same ID can be re-issued sooner. And, when an ID has been re-issued sooner and a stale ID is used to lookup, the chances are high that the ID lookup succeeds.

The ID Manager also permits the reservation of a specific ID and associates the ID with a pointer. ID reservation is commonly used in distributed systems. A typical example can be in a RP, where the LC system and LC have an ID table and the RP wants to maintain a copy of that ID table locally (on the RP). In this case, since the ID Manager on the LC is responsible for allocating an ID, and on the RP, we just want to follow what the LC's ID Manager has done (on the RP we already know the ID that was copied from the LC), so we use `id_reserve()` to allocate the specific ID.

Note If the IDs used for reservations are coming from different ID tables, then both the tables have to be identical. If the allocator ID table is set to “no_randomize_id”, then the reservation ID table has to be set the same as well.

4.12.1 ID Manager API

The ID Manager API functions are listed here:

- [`id_create_new_table\(\)`](#)
- [`id_delete\(\)`](#)
- [`id_destroy_table\(\)`](#)
- [`id_get\(\)`](#)
- [`id_reserve\(\)` \(New in 12.4\)](#)
- [`id_table_set_no_randomize\(\)` \(New in 12.4\)](#)
- [`id_to_ptr\(\)`](#)

See the reference pages in the *Cisco IOS API Reference* for detailed information on these functions.

4.12.2 How to Use the ID Manager

Use the ID Manager to allow clients to avoid passing pointers to actual memory locations as shown in the following steps. In this example, `subsystem_A` has a state that it wishes to protect, and thus acts as a client of the ID manager. Then, `subsystem_B` acts as a client of `subsystem_A`.

- Step 1** The `subsystem_A` must first create an ID Manager table using the `id_create_new_table()` function:

```
id_create_new_table(numTableSlot, numIDsPerSlot, *owner_malloc_name);
```

This function returns a `table_key`, which is the table ID.

- Step 2** Then, `subsystem_A` allocates an ID from the ID Manager table and associates it with a pointer to a piece of `subsystem_A`'s state using the `id_get()` function. This is often done when `subsystem_A` allocates a new piece of state:

```
id_get(void *, id_table_key_t);
```

This function returns an `id_t` type `id`, which is the ID that is to be used instead of a memory pointer.

The `subsystem_A` can now give the returned ID value to its client, `subsystem_B`, as a reference to this state.

- Step 3** When `subsystem_B` later needs to refer to this state within an API of `subsystem_A`, `subsystem_B` provides this ID. `subsystem_A` gets the ID and then passes it using the `id_to_ptr()` function to retrieve its pointer:

```
*id_to_ptr(id_t, id_mgr_ret, id_table_key_t);
```

- Step 4** When `subsystem_A` wants to delete (or otherwise disassociate) its state using the ID, then `subsystem_A` calls the `id_delete()` function to return this ID to the “not in use” state within the handle table:

```
id_delete(id_t, id_table_key_t);
```

Then, `subsystem_A` frees the memory allocated to the state.

Note Once the ID has been deleted, future references to it as in Step 3 (via `id_to_ptr()`) will fail, indicating that the ID has been retired.

- Step 5** When `subsystem_A` is no longer needed, use the `id_destroy_table()` to remove the ID Manager table:

```
id_destroy_table(id_table_key_t);
```

4.12.2.1 How to Get a Sequence of IDs

If the client wants a sequence of IDs (instead of the default randomized IDs), the client can specify `id_table_no_randomize()` to keep the `tableSequence` part of the ID at zero. For example:

```
id_table_no_randomize(remote_rut->ruser_id_table)
```

4.12.2.2 How to Reserve a Specific ID in a Distributed System

If the client wants to reserve a particular ID in a distributed system, the `id_reserve()` function can be used to allocate the specific ID. For example:

```
id_rut = id_reserve(remote_user,
                     remote_rut->ruser_id_table,
                     remote_ru_id & DRMI_USER_MASK);
```

Note The reservation ID table must be exactly the same as the allocator ID table that originally allocated the ID. The `numTableSlot` and `numIDSPerSlot` must be the same, and if `id_table_no_randomize_id()` is called on the allocator ID table, the same must also be done to the reservation ID table.

4.12.3 LRU ID Manager

The Least Recently Used (LRU) ID manager handles a numerical ID space between a specified min and max value (both inclusive in the usable range), allowing the client to request and return IDs.

The allocation strategy of IDs defers reuse for as long as possible. That is, the LRU ID Manager always allocates the least recently used ID.

The LRU ID manager is well-suited for use in an HA environment, because it provides the ability to have an external part mark IDs as allocated, while maintaining the LRU behavior when used for allocating IDs.

The LRU ID manager is used in practice for non-sparse ID spaces of a reasonable size, because it uses an array of list elements internally to store the IDs.

The memory usage is constant and depends on the size of the ID space as follows:

```
if ((max - min) <= 0xFF) {
    n = 1; // sizeof(uint8_t);
} else if ((max - min) <= 0xFFFF) {
    n = 2; // sizeof(uint16_t);
} else {
    n = 4; // sizeof(uint32_t);
}
total_size = 20 + // sizeof(lru_id_mgr_handle)
            n * 2 * (max - min + 1) + // linked list
            ((max - min + 1) >> 3 + 24 - 1) // bitmask
```

For example, an ID space with min=1 and max=2048 uses:

$$20 + 2 * 2 * 2048 + (2048 >> 3) + 24 - 1 = 8491 \text{ bytes}$$

All operations on the ID space except “creation” and “walks” operate in constant time O(1).

The LRU manager was committed using CSCuk60891 and CSCuk60900. See EDCS-529232 for implementation details.

4.13 Shared Information Utility

The Shared Information Utility supports sharing via the shared memory area, using a multiple, versioned view approach such that changes made by a writer are not published and usable by readers in the system until after explicit commit of those changes.

This is shown in Figure 4-10, where the reader is mapped to version 1 of the data and a separate writer process, in a different address space operates on a separate version of the data, writing new data content.

Figure 4-10 Multiple views—pre-commit of new data

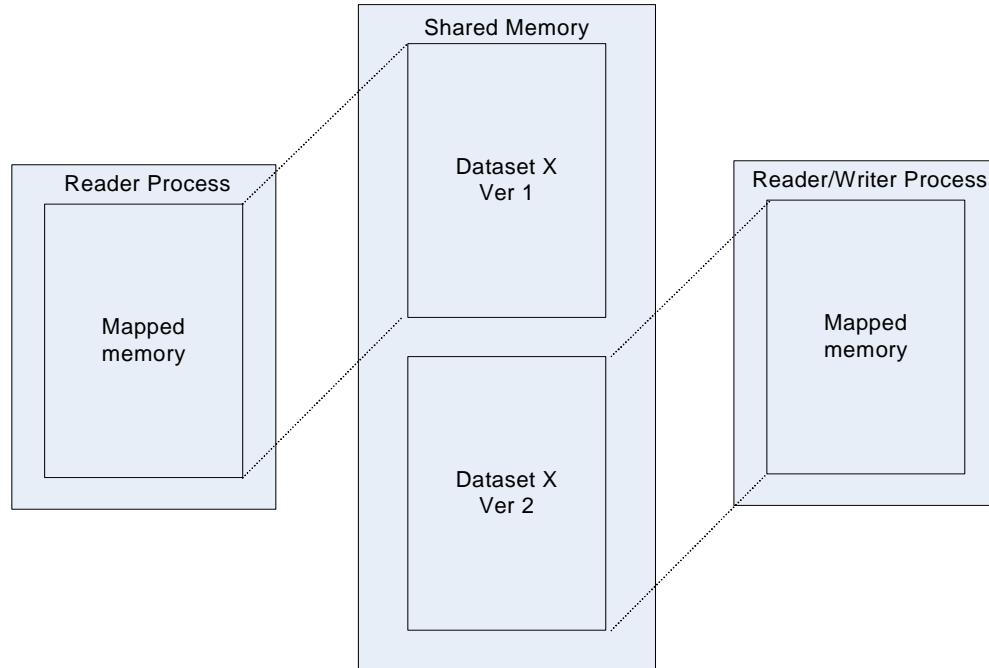
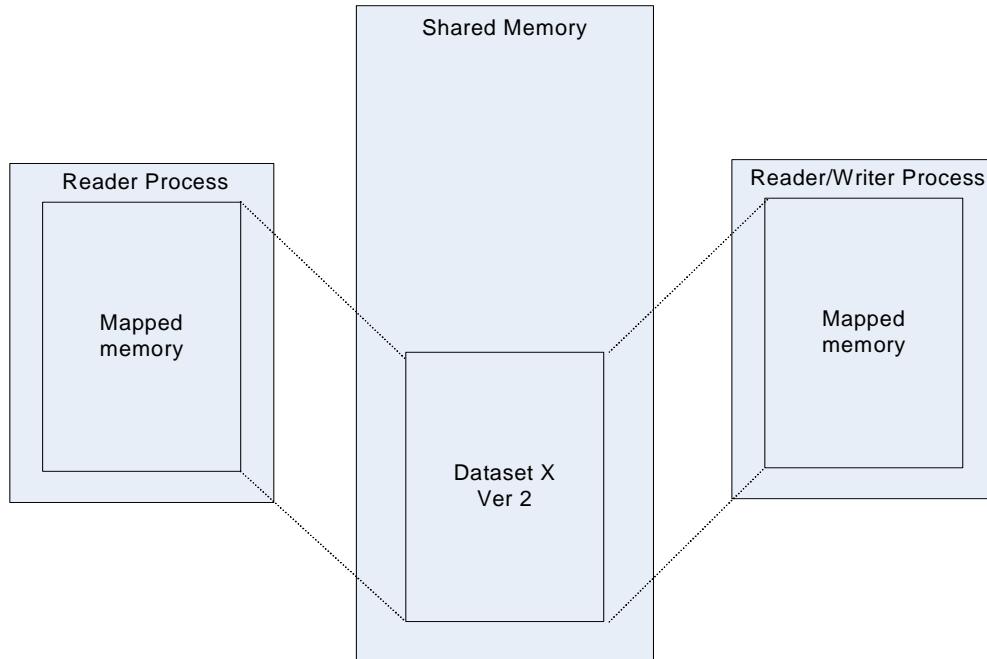


Figure 4-11 shows the state of the views of data after the writer has performed a commit of the new data. Now the reader is mapped to version 2 of the data that was committed by the writer. There are no more processes which need version 1 of the data and it is automatically freed.

Figure 4-11 Multiple views—post-commit of new data



The manner in which updates of a dataset affect a process' view of the data is determined by the mode of operation that is chosen by the process at initialization time. This is described in section 4.13.2 “Reading and Modes of Operation”.

4.13.1 Registration

4.13.1.1 Dataset Registration

Before using the Shared Information Utility an application must register for each of the datasets that it is interested in. Upon registration, using the function `shminfo_dataset_register()`, a handle is returned which is to be used by the application in future reads or writes of information. The name string that is passed as an input parameter to this API routine is used to identify the dataset and needs to be globally unique. In response a unique integer dataset identifier is returned.

A reciprocal routine `shminfo_dataset_deregister()` is used if the application no longer wishes to share this data.

The Shared Information Utility must initialize in the blob before clients in other processes are allowed to register. In order that applications in other processes can know that it is safe to register a dataset that they wish to share, they should check the output of `shminfo_get_is_ready()` and if FALSE then subscribe to the registry service point `shminfo_is_ready` (described in section 4.13.4 “Notifications”) which will indicate that registration can now proceed.

4.13.1.2 Group Registration

In addition to dataset registration, an application may wish to register a grouping of datasets such that consistency is maintained between them during sharing. This is performed using the function `shminfo_group_register()` which returns a handle that is used for future group operations.

A reciprocal function `shminfo_group_deregister()` is used if the application no longer wishes to retain this grouping.

Note that whilst there is no restriction on the number of groups that may be registered, there is a restriction that a dataset may not belong to more than one group.

If a reader wants to pick up a group of datasets then the datasets must be written in a group as well. Note that conversely there is no issue with a writer writing a group of datasets but a reader not registering a corresponding group but instead picking up individual datasets. This may be desirable if one process wants to read groups of datasets and yet another process is only interested in a subset of those datasets.

4.13.2 Reading and Modes of Operation

During registration an application must choose one of three available modes of operation for a dataset. These are described in this section and determine how data is made available to different address spaces.

4.13.2.1 Suspend Update Mode

This is available for IOS-style (i.e. blob or ionized) processes only. In fact this is the only update mode available to IOS-style processes. In this mode, for a dataset, there is only one version of the data that is being looked at in a particular address space. At registration time the reader supplies a pointer to a pointer which is then automatically updated by the infrastructure when new data is committed by a writer process. The reader just needs to dereference its supplied pointer in order get the position of the latest version of the data that it should be operating upon.

For a reader which is in a different address space to the writer, consistency of the same version of data is guaranteed until the time that the current task is taken off the run queue, for example, explicit `process_suspension` or waiting on an event. It is only at this time that the view of the data that the process is looking at may change. During the process of writing, any reads done in the same address space as the writer will obtain the in-progress version of the data.

To show how an application reads the shared data, let's presume that the following has been provided by the application at the time that it registers with the Shared Information Utility. This will then be modified by the infrastructure automatically.

```
static volatile dataset_one_t **ds1_ptr_addr;
```

At any time, in order for the application to obtain a pointer to the version of the data that it is mapped to, it would call the following routine. Note that the returned pointer should not be cached. At process suspension a new version of the data may be mapped into the process' address space, and the old version will then become invalid.

```
static volatile inline dataset_one_t *ds1_data_ptr (void) {
    return (*ds1_ptr_addr);
}
```

So instead of previously referring to "dataset->my_field" it would now refer to "ds1_data_ptr()->myfield" for references to a shared version of the data.

The function `shminfo_get_my_data()` can similarly be used to obtain a pointer to the data as seen by the current thread, for the specified dataset. It also returns the current size and version of the data. Note that the same caveats, as outlined above, apply regarding caching of this obtained pointer.

Note that it is possible for a process using this update mode to skip versions. If a writer commits consecutive versions, N+1 and N+2, in quick succession it is possible that when the reader, currently mapped to version N, comes to update the version that it is mapped to that version N+1 is no longer available and that it will move from version N to version N+2.

4.13.2.2 Immediate Update Mode

This mode is available to POSIX processes only. It is identical in behavior to the Suspend Update mode except that availability of new versions to a reader is not delayed until process suspension. It becomes available as soon as the notification of the new version is handled by the process.

In this mode, reading of the data is done in the same manner as that described in the Suspend Update mode, that is, the application simply needs to dereference the pointer that it supplied during registration.

4.13.2.3 Application Update Mode

This mode is available to POSIX processes only. In this mode there may be multiple versions looked at by the different threads of the process. Each thread may also look at more than one version of a dataset. The approach used to read data, as described in the two previous modes, where the infrastructure automatically updates the version of data that the application should read, is not applicable to this mode. Instead the application must make an explicit call to lock in the version of data that it wishes to look at. This is done via calling `shminfo_reader_lock_version()`. This version is guaranteed to remain current until this version of the data is explicitly released via calling `shminfo_reader_unlock_version()`. It is up to the application to maintain consistency of data between threads (by explicitly causing each thread to lock to the same version of the data) if that is important. Conversely, the application may choose to include version information in any inter-thread messages if necessary.

Whilst the function `shminfo_reader_lock_version()` will return a pointer to the data which the application should use for reading the data, it may not be convenient or desirable to pass this pointer through the application code, particularly if it involves several levels deep of function calls. Hence `shminfo_get_my_data()` is provided and can be used by any thread at any time, to obtain a pointer to the view of the data that it was previously locked to.

Note that calls to `shminfo_reader_lock_version()` are not required in this mode if new views want to be obtained for not just one but rather a group of datasets. In this case `shminfo_reader_lock_group()` is called and the latest versions of all datasets that are defined in the specified group are locked in until the reciprocal API `shminfo_reader_unlock_group()` is called.

In this mode the application will want to subscribe with the registry service point, `shminfo_data_ver_update()`, in order to obtain notifications of newly committed versions of the data.

4.13.3 Writing

4.13.3.1 Obtaining New Write View

When an application that has write access wants to change parts of a dataset, it needs to call `shminfo_writer_data_get_writeptr()` in order to obtain a new view of the data. A pointer is returned to the address at which the application should write this information. A complete copy of the most current version of the data is handed to the application and hence it only needs to write those fields which it wishes to change.

Note that the newly updated information is not available to other processes until a commit of the data has been performed.

Note that there is only one in-progress write view of the data at any one time in a process. If `shminfo_writer_data_get_writeptr()` is called again before the data is committed then the application will obtain a pointer to the same in progress write view.

4.13.3.2 Commit

For both IOS style and POSIX processes, when an application is ready to publish its modified data to other address spaces, it must call `shminfo_writer_data_committable()`. Note that the actual commit may be delayed temporarily by the infrastructure in order to ensure consistency of this dataset or the group (if the dataset belongs to one).

4.13.3.3 Abort

If an application has previously obtained a write view of the data and now finds due to an application error that it cannot commit the data, it should call `shminfo_writer_data_abort()` in order to avoid the commit and also in order to release the write view.

4.13.3.4 Callbacks

Routines which modify data often call notification code via callbacks or more commonly registry calls in order to inform interested parties of a change in the data. In a multiple address space system, where the changed data is not instantaneously available in address spaces outside that of the writer, such notifications constitute information leakage. This is dangerous since a different process can then be operating on data which is inconsistent with the information that it received via the notification. In order to avoid such premature publishing of new data, `shminfo_writer_commit_cb_add()` should be used by an application to register a callback that will be called after the next data commit (only). Note that this is generally only relevant for notifications to other processes. Notifications within a process generally should still be made immediately since reads within a writer process are made from the write view which includes newly modified and yet to be committed information.

4.13.3.5 Enabling and Disabling Commit

For consistency reasons it may be necessary to temporarily explicitly disable a data commit. Such an example would be when there are two set API routines which need to be called consecutively but it is the desire to publish this data only once. Another example relates to group consistency where it may be necessary to publish a change in two or more datasets in the group at the same time.

The functions `shminfo_writer_disable_commit()` and `shminfo_writer_enable_commit()` are used by the application in order to perform this commit disabling and enabling.

4.13.3.6 Multiple Writers

Note Multiple writers, in different processes, are not supported in the first release.

For each process there is a maximum of one in progress write view per dataset. If there is a writer that is currently writing a new data view and a second application in the same process requests a new data buffer then it will be returned a pointer to the same buffer that the first application is in the process of writing. Consequently it is highly recommended to have no more than one writer application for a dataset, in the same address space, in order to prevent one application overwriting another's write changes.

If multiple processes need to make a change to the same dataset it is often preferable that one process has the sole responsibility of making changes to that dataset and that other processes request that writer process to make the changes on behalf of them. This ensures that the writes are serialized in a controlled fashion. If this approach is not taken, and multiple processes are allowed to directly commit new write views of the data, then the following sequence must be done.

- 1 Writer process gets a new write view of the data
- 2 Application makes changes to the data
- 3 Attempt to commit data is performed upon call of `shminfo_writer_data_committable()` or `shminfo_writer_enable_commit()`. If either of these fail it indicates that another process committed the same dataset while we were making changes to our write view. In this case the current write view is automatically released and the application must return to step 1. Note that it is possible that the newly written data has changed the view of the application that is trying to do the write and hence it must reevaluate and potentially may need to change the content of the data that it needs to write.

In summary, the following are the different ways that multiple writers could be supported in order of decreasing desirability:

- 1 Only one writer in the system

If other processes want to create a new data view then they should request the designated writer to make the change on their behalf. This is the safest and highly recommended approach.

- 2 Writers in multiple processes with global lock

In this case a global system lock, operating on a higher level than this utility, is used to prevent any writer from requesting a new write view if there is another already in the process of creating a new view. Note that in this case a potential writer may be blocked until there are no writes in progress and the newly written data, changed by another application, may now have an effect of what the new writer should write.

- 3 Multiple writers but without lock

In this case the above sequence needs to be undergone. As with 2) this may lead to excessively complex application code since it needs to support the ability to retry a write, and potentially change what needs to be written based on data that was just written.

4.13.4 Notifications

In order for a process to be informed of a change in the dataset that it is interested in, an application should register with the service point `shminfo_data_ver_update()`. In the case of an IOS-style process this registry service point is invoked directly after the new view of the dataset is mapped into that address space. In the case of a POSIX process using the application update mode, the dataset is not automatically mapped and hence subscription to this registry will commonly be used in order to notify the application that a new version of the data is available and that the application can map this as soon as it is appropriate for the process.

The following service points are defined in the file `shminfo_registry.reg`. Include the header `shminfo_registry.h` if the application needs to subscribe to this registry.

```

DEFINE shminfo_data_ver_update
/*
 * Used by clients of the Shared Information Utility infrastructure to be
 * notified of a newly available version of the specified data.
 */
CASE_LIST
void
shminfo_ver_t ver | IN
SHMINFO_DATA_MAX
shminfo_data_type_t type
END

DEFINE shminfo_is_ready REMOTE, DONTWAIT
/*
 * This is used to signify to a prospective client that it can now safely
 * register with the Shared Information Utility. It is invoked when
 * the blob has initialized this utility.
*/
LIST
void
-
END

```

Note that the above is intentionally not marked as being a REMOTE service point. This is because it is the Shared Information Utility infrastructure in each process that invokes this registry. As a result, if there are several interested parties in the same process, only one remote registry call is required, instead of one remote call per subscriber.

4.13.5 Shared Information Utility API

API reference pages are available for the following `shminfo_*`() functions (*New in 12.2SX*):

- [1 `shminfo_dataset_deregister\(\)`](#)
- [2 `shminfo_dataset_register\(\)`](#)
- [3 `shminfo_get_is_ready\(\)`](#)
- [4 `shminfo_get_my_data\(\)`](#)
- [5 `shminfo_group_deregister\(\)`](#)
- [6 `shminfo_group_register\(\)`](#)
- [7 `shminfo_reader_lock_group\(\)`](#)
- [8 `shminfo_reader_lock_version\(\)`](#)
- [9 `shminfo_reader_unlock_group\(\)`](#)
- [10 `shminfo_reader_unlock_version\(\)`](#)
- [11 `shminfo_writer_commit_cb_add\(\)`](#)
- [12 `shminfo_writer_data_abort\(\)`](#)
- [13 `shminfo_writer_data_committable\(\)`](#)
- [14 `shminfo_writer_data_get_writeptr\(\)`](#)
- [15 `shminfo_writer_disable_dataset_commit\(\)`](#)
- [16 `shminfo_writer_disable_group_commit\(\)`](#)
- [17 `shminfo_writer_enable_dataset_commit\(\)`](#)
- [18 `shminfo_writer_enable_group_commit\(\)`](#)

Pools, Buffers, and Particles

The information in this chapter is divided into the following sections:

- Section 5.1, “Buffer Management: Overview”
- Section 5.2, “Generic Pool Management”
- Section 5.3, “Packet Buffer Management”
- Section 5.4, “Particle-based Buffer Management”
- Section 5.5, “Useful Buffer Commands”

5.1 Buffer Management: Overview

The main 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 is the maximum size. In fact, frames as small as 64 bytes are common on Ethernet. For the underlying packet support code to be efficient, it must handle a wide range of possible sizes without wasting memory, while at the same time not complicating the underlying code.

The Cisco IOS packet and particle-based buffer management components use *buffers* to manage the memory in which to hold and manipulate packets. Buffers are allocated using the Cisco IOS memory management code (functions such as `malloc()`, `chunk_malloc()`, and their variations), which cannot allocate memory at interrupt level where packet manipulation often takes place. As a result, buffers are preallocated at initialization time, and made available to system components at both process and interrupt levels. The preallocated buffers are maintained in *pools*.

Pools provide a general management mechanism for a group of items that have similar characteristics. When items in a pool are available to any component in a Cisco IOS system, the pool is a *public pool*. Most platforms set up several public pools of packet buffers used primarily for packets originating on the local device. For forwarding or switching packets, a particular interface can set up and use *private pools* for its buffers, for which access is restricted only to that interface. This prevents traffic overloads on other interfaces from affecting the interface’s resource availability. Pool clients can also allocate a *pool cache* to allow quicker item acquisition for performance-critical operations. Pool caches are usually only associated with private pools.

Currently, the pool management code is used by the following two integrated packet buffer management services:

- Packet Buffer Services, for handling contiguous *packet buffers*.
- Particle Services, for handling packet data in discontiguous buffers or *particles*, for scatter-gather DMA packet manipulation.

Note Much of the Cisco IOS process-level forwarding code is still particle-unaware and only handles data presented in contiguous packet buffers. Interface driver switching code generally runs at interrupt-level, and on many platforms implements particle-based packet buffer handling. The packet and particle-based buffer APIs offer functions to coalesce particle-based packets into contiguous packet buffers, if necessary, for particle-unaware upper level handling.

The pool management code supports setting up *dynamic pools*, which allow the number of items within them to grow and shrink on demand. For example, the Packet Buffer Services use dynamic pools to allow buffer resources to adapt to match the current packet load.

Using pools for packet and particle-based buffer management allows resources to be handled effectively under various code execution constraints. Preallocating pools of buffers allows memory to be available for handling packets at both process and interrupt levels, and provides more efficient memory usage and access time for handling different MTU sizes.

5.1.1 Terminology

buffer

An area in Cisco IOS memory that holds a packet while it is manipulated by a network device. Internally, the generic pool management code often refers to pool items as “buffers” because that is its primary application: buffer management.

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.

contiguous DMA

The basic packet manipulation method in which an entire packet is stored in a single buffer. This method is less memory efficient than scatter-gather DMA (packet buffers).

dynamic pool

A pool for which the number of items it contains can change over time to reflect the current needs. 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.

header pool

A pool of Cisco IOS packet header items only, which are defined by the `paktype` structure. There are no packet data buffers associated directly with each packet header item in the pool.

input queue of an interface

Cisco IOS software refers to the count of the number of packets in the system that are associated with a particular interface as that interface's *input queue*, although this structure is not actually represented internally as a queue. Packets that either came into the system from an interface or had ownership transferred to an interface are considered to be *charged* to that interface and are credited to the "input queue" count, which is maintained in the `hwidb` for the interface.

interface or network interface

The hardware or firmware that provides a mechanism to move packets between the network link and storage (memory) on the network device.

interface driver or network interface driver

The software component of a Cisco IOS image that controls the interface.

packet

Data that is either to be transported over a network (forwarded or switched), originates in the local network device (such as ARP messages exchanged with neighboring devices), or is "consumed" on the local network device (such as SNMP control messages). Packet and frame are sometimes interchangeably used, although in this document, frame usually refers to DLL data encapsulated in a packet.

packet buffer

A Cisco IOS method to represent and manipulate packets using two elements, a structure to hold the packet data and a `paktype` header structure that maintains significant information about and references into the packet data.

packet buffer pool

A pool of packet buffers. "NAME packet buffer pool" refers to one of the 6 different size pools named for their item size that are created during system initialization for use by any system component. These packet buffers are used for jobs such as console logging, manipulating packets that originate on the local network device (for example, ARP inquiries), and fallback packet buffer allocation when an interface's private pool has no available buffers.

Packet Buffer Services

The Cisco IOS code that manages packet and particle-based buffers using the Cisco IOS pool management code.

particle

A Cisco IOS storage element that holds a portion of a packet implemented by a scatter-gather DMA (packet buffers) method. A particle consists of a particle header structure, `particletype`, and a particle data buffer, and represents part of a contiguous packet. Particles are connected to a packet buffer header in a chain of evenly-sized blocks that are smaller than the maximum MTU-size packet of an interface, and represent the entire packet with "scattered" blocks located randomly in memory.

particle-based packet

A particle chain that represents an equivalent, contiguous packet, attached to a packet buffer header.

particle chain

A list of particles linked together by fields in the `particletype` header that represents part or all of a particle-based packet.

particle pool

A pool that contains particles and implements scatter-gather DMA (packet buffers).

pool

A container in which to hold and manage similar items. The container structure holds items that are available or not currently in use; items that are in use maintain a reference back to the container so they can be returned to the pool when they are no longer in use. Data structures such as buffers are managed in pools. The Packet Buffer and Particle Services are two main IOS buffer pool clients, and maintain packet buffers or particles in pools.

pool cache

An array of pointers to pool items that allows faster retrieval of available items. For buffer pools, they are generally used only with private pools by interface driver code.

pool client

Code that uses the Cisco IOS pool management code directly and calls pool API functions to manage items, for example, to provide a pool-based service such as the Packet Buffer or Particle Services. Also can refer to a second level of pool clients: clients who use a pool-based service (a pool client service) by calling pool wrapper functions, and thus are indirectly clients of generic pool management functions.

pool client services

Pool clients such as the Packet Buffer or Particle Services, that provide a pool-based service through wrappers around generic pool API functions.

private pool

A pool of items to which access is restricted. Private pools are generally used to provide better resource control but require more management overhead.

public pool

A pool of items available to all Cisco IOS components in the system. Public pools provide memory efficiency at the cost of potential resource contention when multiple components share access to pool items. For Packet Buffer Services, for example, typical Cisco IOS platforms allocate several public pools of packet buffers of different sizes at initialization time.

scatter-gather DMA (packet buffers)

A packet manipulation feature provided by the network interface and Cisco IOS driver software that allows a packet to be read from or stored in multiple buffers, and reconstructed as a contiguous packet for transmission. This feature improves efficiency of memory usage for packet manipulation.

static pool

A pool for which the number of items it contains remains the same.

5.1.2 Packet Buffers

The Cisco IOS Packet Buffer Services use the generic pool code to manage data in contiguous buffers for receiving and transmitting network interface packets. These services provide the following features to help developers solve packet transfer challenges across multiple platforms:

- Use IOMEM (isolated-access Input/Output Memory for a network interface) for packet data when available, and processor memory for packet information.
- Support different MTU sizes based on interface types.
- Maintain a set of preallocated *public* (shared) packet buffer pools for use by any image component.

- Support creation and maintenance of additional public or *private* (exclusive) packet buffer pools for a particular interface.
- Allow allocation of private buffer pool caches for optimal packet availability.
- Allow fallback buffer allocation when a preferred packet buffer pool is exhausted.
- Support buffer locking, packet duplication, and interface input and output queuing functions.

5.1.3 Particle-based Buffers

Particles are a more recently-developed buffer mechanism in Cisco IOS packet buffer management. Traditionally, the Cisco IOS protocol and application code uses a packet buffer structure that includes a contiguous area of memory for the frame data. To permit drivers to support *scatter-gather DMA*, the Cisco IOS software also allows frame data to be composed of individual blocks called *particles*. With scatter-gather DMA (Direct Memory Access), a DMA of a contiguous block of data is spread across multiple smaller pieces of memory. For example, a 1500-byte frame might be contained in three 500-byte pieces. Using particles, or *particle-based buffers*, can result in 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.

Currently, many Cisco IOS platforms use particles to implement fast-switching in interrupt-level interface driver, encapsulation, and forwarding code. In general, adding fast-switching support to a Cisco IOS component now implies use of particle-based buffers. However, most Cisco IOS process-level forwarding code (and other process-level packet handling code) was created before particles were developed, and handles only contiguous packet buffers. As a result, particle-based packets must be coalesced into a contiguous packet buffer before being passed to particle-unaware upper-level processing code. This can impact process-level switching performance, and the use of particles within a platform's driver structure must be carefully designed and examined. The use of particles is emerging in the Cisco IOS code base; before using particles in any code, seek design advice from senior Cisco IOS development engineers.

Besides allowing scatter-gather DMA, particle-based buffer management is based upon the existing pool management code, and provides the following features for particles:

- Use IOMEM (isolated-access Input/Output Memory for a network interface) for particle data, when available, and processor memory for particle header information.
- Allow creation of *public* (shared) or private particle pools and private particle pool caches, and assigning a fallback particle pool when a preferred pool is exhausted.
- Support allocating particles from and returning particles to particle pools.
- Allow setup and manipulation of particles on a particle chain (also called particle queue) that make up a whole packet.
- Support particle cloning, which assigns two copies of a particle's header to point to the same particle data buffer, creating two logical copies of a packet without duplicating the data (for example, for multicasting).
- Allow moving particle chains to a new packet header, called *reparenting*.
- Support converting particle-based buffers to a contiguous packet.

Note Cisco IOS pool, packet buffer, and particle development questions can be directed to the interest-os@cisco.com and os-infra-team@cisco.com aliases.

5.2 Generic Pool Management

The Cisco IOS generic pool operations are built upon the IOS list management services, and meet the challenges of managing client resources such as packet data buffers. These operations are flexible enough to support resource requirements of different clients, without requiring clients to duplicate complicated management operations. The generic pool implementation can meet changing client demands for the resources in a pool by growing and trimming resources while preserving the integrity of the pool against context changes between process and interrupt levels.

The Cisco IOS packet buffer and particle pools are structured on top of this generic pool framework, and each provide API functions for optimal handling of packet data for different platforms and interfaces.

5.2.1 Generic Pool Operation and Characteristics

Cisco IOS systems generally create resource pools and populate them with items at process level, typically during system initialization. During system operation, pool clients acquire items from the pool for use, and return them to the pool when they are no longer needed. When a client no longer requires the resources in a pool, the client can destroy the pool; the pool container and all the items it contains are returned to the memory manager for freeing the memory space. The pool management code checks to make sure all users of a pool item have indicated they are finished with it before releasing the item, and assures that all items have been released before destroying the pool.

Pools maintain characteristics such as the following:

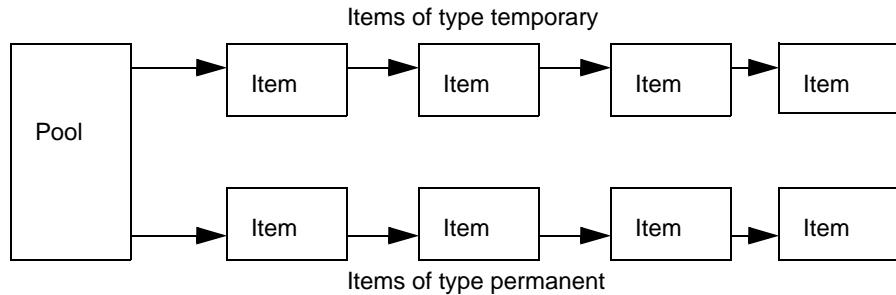
- Item size: Items in a pool are of the same size; the item size can vary from pool to pool.
- Access to items: A pool can be universally accessible (public) or have restricted access (private)
- Number of items contained: A pool can be defined to contain a fixed number of items (static) or the number of items associated with it can grow and shrink over time (dynamic) based on client demand for its resources. Dynamic pools support modifying the item number parameters that specify the minimum and maximums to which the pool conforms.
- Caches: A pool can maintain an array of its items to allow faster acquisition of an available item.
- Function vectors: Each pool maintains a table of function vectors to define and allow customization of pool item use, for example, to create a new item for the pool, destroy a pool item and return its memory to the memory manager, obtain (or get) a item from the pool, and return an item to the pool when it is no longer being used.

The remaining sections describe the pool structure and how these characteristics are implemented in detail.

5.2.2 Pool Structure

In the basic pool architecture (see Figure 5-1), a pool is described by a `pooltype` structure. From this structure, two queues of pooled items are constructed, in which one queue contains permanent items and the other one contains temporary items (see subsection 5.2.6, “Permanent and Temporary Items”). All items in the queue are available and ready to be consumed by pool users. The pool items are threaded together using the singly-linked list (queue) support, in which the first member of each data item on the queue points to the next item on the queue. The pools themselves are usually threaded together using Cisco IOS List Manager Services. (Queues and list manager operations are discussed in “Chapter 22, “Queues and Lists”.”)

Figure 5-1 Structure of a Pool



5.2.3 Pool Group and Item Size

Pool clients identify a pool by the following main parameters:

- A pool group number
- A size associated with all items stored in the pool

Pool group numbers allow partitioning or partial association among pools, and they can allow a pool creator to restrict access to a pool. The primary application of group numbers is shown in the Packet Buffer Services (see subsection 5.3.3.1, “Public and Private Packet Buffer Pools”), where the pool group number 0 is reserved for public packet buffer pools, and a unique, non-zero pool group ID is acquired for allocating private packet buffer pools.

Private or non-zero pool groups are usually made up of pools that all have the same item size. The public pool group, group 0, generally has several pools associated with it. A pool in pool group 0 can be associated with a different item size than another pool in the group. In addition, the packet buffer and particle pool APIs internally consider applicable encapsulation and trailer padding in pool item sizes when creating a pool, to allows expansion at different protocol layers without having to move the item data. Externally, pool users refer to a data size exclusive of the padding.

5.2.4 Pool Lists

When a pool is created, it is associated with a list, which is a pointer to a preallocated `list_header` structure. Pool services often use lists of pools to manage access to pool items with certain shared characteristics. For example, the Packet Buffer Services associate all public pools with one list, and private pools with another. Upon creation, a pool is inserted into the associated list in ascending order by group number and, within groups, by item creation size.

5.2.5 Static and Dynamic Pools

Pools are classified as being either *static* or *dynamic* when they are created. Static pools do not change the number of items they contain during operation. An attempt to get an item from a static pool fails if there are no free items in the pool. With dynamic pools, the pool attempts to meet the resource demands of its clients. The pool manager attempts to add items when the pool is empty, if it can do so within the context of the client requesting an item. Otherwise, a critical priority background task is scheduled to run at the next available processor interval to replenish the pool.

5.2.6 Permanent and Temporary Items

Items in a pool are classified as being either *permanent* or *temporary*. Permanent items are always in a pool and are not destroyed unless the parameter defining the number of permanent items is adjusted, or the entire pool is destroyed. 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 when the pool is populated. These items can be removed from the pool if the number of free items rises above the maximum specified.

5.2.7 Create a Pool

First, because the generic pool management structure is built upon on the Cisco IOS list and memory management services, creating a pool requires prior allocation of list and memory pool resources. Typical implementations of particular pool client services, such as those for managing packet buffers and particles, provide wrappers around the generic `pool_create()` function to create pools with the characteristics they support. The system initialization functions and wrappers for these services preallocate the list and memory resources they require. Because the primary pool management implementations in Cisco IOS systems are for packet and particle buffer management, you usually would not call `pool_create()` directly. Section 5.3, “Packet Buffer Management”, and section 5.4, “Particle-based Buffer Management”, give details on packet buffer and particle pool management functions.

For those who need to understand more about what is involved in setting up a pool, this section describes generic pool creation, setting up the prior pool support structures, and defining pool characteristics.

To create a pool, call the `pool_create()` function. This function creates only the pool container structure for items whose size is specified by the `size` parameter. No items are placed in the pool until it is populated and its characteristics defined with the `pool_adjust()` function. Then the pool container is inserted into the pool list associated with its group number, in order of increasing item size within the list.

```
pooltype *pool_create (char *name, int group, int size, uint flags,
                      mempool *mempool, list_header *list,
                      pool_item_vectors *item);
```

Table 5-1 describes the `pool_create()` parameters. The subsections following the table include more details about prior operations required for some parameters, as referenced in the table.

Table 5-1 pool_create() Parameters

Parameter	Description
<code>name</code>	An identifier string assigned to the pool for display purposes.
<code>group</code>	Specify 0 for a public pool, or specify a private pool group number obtained by calling the <code>pool_create_group()</code> function.
<code>size</code>	The size in bytes of items this pool will contain.
<code>flags</code>	Values that determine some pool characteristics. See subsection 5.2.7.2, “Pool Flags”.
<code>mempool</code>	A pointer to the memory pool from which the pool’s item create function vector should allocate and deallocate space for items (see section on function vectors). Pool client services usually interpret a NULL pointer as the default memory pool, which, for packet and particle buffers, is IOMEM (if available on the platform) or local system memory.

Table 5-1 pool_create() Parameters (continued)

Parameter	Description
<i>list</i>	A pointer to a List Manager list structure in which the pool is maintained. See subsection 5.2.7.3, “Pool List Structure”.
<i>item</i>	A pointer to a pool item function vector block specified by the <i>item</i> parameter. These vectors are defined by pool client services to perform appropriate pool item operations.

5.2.7.1 Pool Group Number and Creating Private Pools

A *pool group* is a set of pools that are linked together by a common identifying *pool group number*.

By convention, Cisco IOS pool clients use pool group number 0 as the *public pool* group, for which any components in a Cisco IOS system have access. The item size characteristic of a pool usually varies from pool to pool in the public pool group. Pool client services can implement a seamless memory-efficient item retrieval function that finds and returns a pool in the public group with an available item that most closely matches the requested size. The Packet Buffer Services *getbuffer()* function provides this capability for retrieving an available packet buffer of a particular size.

A *private pool* is a pool to which access is restricted only to pool clients who create or share knowledge of its (non-zero) pool group number with selected components. Private pools allow clients to more closely control their pool resources. By convention, pool clients use a unique pool group number to identify a private pool. The generic pool function, *pool_create_group()*, returns a unique integer that can be used as a private pool group number.

Follow these steps to create a private pool:

Step 1 Obtain a unique group number by calling *pool_create_group()*:

```
int pool_create_group(void);
```

Step 2 Pass the unique pool group number to *pool_create()* or a wrapper pool creation function such as *pak_pool_create()*, as shown in the following example of creating a private packet buffer pool container:

```
/*
 * nip_init_private_pool:
 * Init NIP private buffer pool with cache
 */
void nip_init_private_pool (void)
{
    int pool_group;
    nip_buffer_pool = NULL;
    pool_group = pool_create_group();

    nip_buffer_pool = pak_pool_create("DSIP", pool_group,
                                      NIP_POOL_BUFSIZE,
                                      POOL_DEFAULT_FLAGS, NULL);

    if (!nip_buffer_pool) {
        errmsg(&msgsym(PRIVPOOL, DSIP));
        return;
    }
}
```

Remember that pool creation simply sets up the pool container. The pool must later be populated with items.

5.2.7.2 Pool Flags

Table 5-2 lists values for the *flags* parameter to the `pool_create()` function. Some values such as `POOL_CACHE` and `POOL_DYNAMIC` are interpreted by the generic pool management functions, while others influence how pool wrapper functions will handle management of the particular pools they support. Refer to the flags descriptions in each pool client service's pool creation wrapper (such as `pak_pool_create()` and `particle_pool_create()`), for more details.

Table 5-2 Pool flags Definitions

Value	Description
<code>POOL_DYNAMIC</code>	If set, the number of items in the pool can be adjusted dynamically by a background periodic call to the <code>pool_prune()</code> function. New items are allocated and destroyed using the <code>create</code> and <code>destroy</code> function vectors in the <code>item</code> parameter (see the information on item function vectors later in this section). If this flag is clear, the pool is static and the number of items remains the same until the pool is destroyed or reconfigured using the <code>pool_adjust()</code> function.
<code>POOL_SANITY</code>	If set, items are considered for memory pool sanity checks, when enabled with the <code>debug sanity</code> command. If this flag is clear, no memory pool sanity checks will be performed.
<code>POOL_CACHE</code>	If set, a cache has been associated with the pool. If this flag is clear, the pool is not using a cache.
<code>POOL_PUBLIC_DUPLICATE</code>	If set, pool operations involving duplication of pool items should always use public pools to acquire the new item when copying an item from this pool. If this flag is clear, pool operations should instead try to acquire the new item from this pool itself.
<code>POOL_INVISIBLE</code>	If set, information about this pool is not displayed by pool display commands. Otherwise, the information should be displayed.

5.2.7.3 Pool List Structure

Pool clients must first create a list structure in which its pool container will be maintained, and provide a pointer to the list as the *list* parameter of the `pool_create()` function. The newly created pool is added to that list. A pool client service usually sets the underlying list structures and creates the pools it will manage in the service's initialization function. Refer to Chapter 22, "Queues and Lists", for details on using List Manager API functions.

The following example code from the `pak_pool_init()` function illustrates a pool client service initializing the underlying list structures for the set of public packet buffer pools that it makes available to all packet buffer clients in the system:

```

/*
 * Initialize buffer pool lists
 */
list_create(&publicpoolQ, 0, "Public Pools", LIST_FLAGS_INTERRUPT_SAFE);
list_create(&privatepoolQ, 0, "Private Pools", LIST_FLAGS_INTERRUPT_SAFE);
list_create(&headerpoolQ, 0, "Header Pools", LIST_FLAGS_INTERRUPT_SAFE);

```

```

list_set_info(&publicpoolQ, bufferpool_list_info);
list_set_info(&privatepoolQ, bufferpool_list_info);
list_set_info(&headerpoolQ, bufferpool_list_info);

/*
 * Create a free list for packet headers
 */
mempool_add_free_list(MEMPOOL_CLASS_LOCAL, sizeof(paktype));

small = pak_pool_create("Small", POOL_GROUP_PUBLIC,
                        SMALldata, POOL_DEFAULT_FLAGS, NULL);
middle = pak_pool_create("Middle", POOL_GROUP_PUBLIC,
                         MEDDATA, POOL_DEFAULT_FLAGS, NULL);
big = pak_pool_create("Big", POOL_GROUP_PUBLIC,
                      BIGDATA, POOL_DEFAULT_FLAGS, NULL);
verybig = pak_pool_create("VeryBig", POOL_GROUP_PUBLIC,
                          VERYBIGDATA, POOL_DEFAULT_FLAGS, NULL);
large = pak_pool_create("Large", POOL_GROUP_PUBLIC,
                       LARGEDATA, POOL_DEFAULT_FLAGS, NULL);
huge = pak_pool_create("Huge", POOL_GROUP_PUBLIC,
                      DEF_HUGEDATA, POOL_DEFAULT_FLAGS, NULL);

```

5.2.7.4 Pool Item Function Vectors

The key to a pool's operational characteristics is the pool item function vector block specified by the *item* parameter to the `pool_create()` function. These vectors are defined by pool client services to perform appropriate pool item operations corresponding to their vector names shown in Table 5-3.

Note Function vector blocks vary slightly across release trains, so refer to the "pool.h" include file for the expected `pool_item_vectors` for a particular code base.

Table 5-3 **Pool Item Vectors**

Vector	Description
create	Creates a new pool item and returns a pointer to it.
destroy	Removes an item from the pool and returns the space to the memory pool from which it was allocated.
get	Obtains a pool item from the pool's queue of available items. This vector can also create a new item if the pool is dynamic, and the number of available items is below the limit defined by the <code>pooltype</code> structure's <code>minfree</code> member. The <code>minfree</code> value is set from the <code>mincount</code> parameter of the <code>pool_adjust()</code> function when it populates or adjusts the pool.
ret	Returns a pool item to the pool's queue of available items. This vector can also destroy an item if the number of available items is above the limit defined by the <code>pooltype</code> structure's <code>maxfree</code> member. The <code>maxfree</code> value is set from the <code>maxcount</code> parameter of the <code>pool_adjust()</code> function when it populates or adjusts the pool.
status	Fills in a status block for an item. An item's status consists of its state (temporary or permanent) and its age. (<i>Deprecated in releases after 12.4</i>)

Table 5-3 Pool Item Vectors (continued)

Vector	Description
lock	
unlock	
reset	
refcount	
validate	Validates a pool item for debugging support.

The item vector structure and vector function prototypes are as follows:

```

typedef void * (*pool_item_create_t)(pooltype *pool, pool_item_type type);
typedef void (*pool_item_destroy_t)(pooltype *pool, void *item);
typedef void * (*pool_item_get_t)(pooltype *pool);
typedef void (*pool_item_ret_t)(pooltype *pool, void *item);
typedef void (*pool_item_status_t)(pooltype *pool,
                                  void *item,
                                  pool_item_status *status);

typedef boolean (*pool_item_validate_t)(pooltype *pool, void *item);
typedef void (*pool_item_lock_t)(void *item);
typedef void (*pool_item_unlock_t)(void *item);
typedef void (*pool_item_reset_t)(void *item);
typedef int (*pool_item_refcount_t)(void *item);

typedef struct pool_item_vectors_ {
    pool_item_create_t    create;
    pool_item_destroy_t   destroy;
    pool_item_get_t       get;
    pool_item_ret_t       ret;
    pool_item_lock_t      lock;
    pool_item_unlock_t    unlock;
    pool_item_reset_t     reset;
    pool_item_refcount_t  refcount;
    pool_item_validate_t  validate;
} pool_item_vectors;

```

5.2.8 Populate or Adjust a Pool

A newly-created pool must be populated with items before use. To fill a pool with items and establish the operating parameters for it, call the `pool_adjust()` function. This function can also be used to adjust the contents and change the operating parameters of a pool sometime later, for example, in response to reconfiguration requests to handle performance issues.

```
void pool_adjust(pooltype *pool, int mincount, int maxcount, int permcount,
                 boolean default);
```

This function sets the requested minimum and maximum number of free items, and the number of permanent items in the pool. Then it allocates free items to satisfy these parameters using the pool item `create` function vector. At least `permcount` items will be created that are permanently in the pool, and for dynamic pools, additional temporary items added to satisfy `mincount`, as described next.

5.2.8.1 Pool Parameters in Dynamic Pools

Pools specified as dynamic at creation can adjust the number of available items to meet the current demand. When many items are in use, new *temporary* items can be created to meet the demand. When enough items have been returned, the available item count can be reduced by destroying temporary items. The generic pool manager works with pool client services to attempt to keep the number of available items within a specified range.

Table 5-4 describes the parameters to the `pool_adjust()` function that establish the desired limits for the number of available items in a dynamic pool. The table indicates the `pooltype` (pool container structure) member in which each parameter value is stored for later checking.

Table 5-4 Dynamic Pool Parameters in `pool_adjust()` Function

Parameter	Description	pooltype Member
<code>permcount</code>	Number of items that should remain in the pool.	<code>permtarget</code>
<code>mincount</code>	Minimum number of available items that the pool management code should try to maintain in the pool.	<code>minfree</code>
<code>maxcount</code>	Maximum number of available items that the pool management code should try to maintain in the pool.	<code>maxfree</code>

Generic pool management relies on memory resources managed by the Memory Pool Manager (see section 4.1.3, “Memory Pools, Memory Pool Manager, and Free Lists”). One of the main restrictions of the memory manager is that memory obtained using `malloc()` and related functions can only be allocated at process level, so the generic pool code cannot create new items while running within an interrupt handler. Therefore, a critical process level task can be scheduled that attempts to keep the number of items specified by `mincount` in the pool. If an item is requested from a pool and the free item count is below the minimum value, the pool code attempts to grow the pool to the minimum free count by allocating and adding new, temporary items. The `mincount` value therefore represents the maximum number of items that the pool has reserved for interrupt-level requests.

When an item is requested at process level, the pool’s item `get` function vector can check the item free list count against `minfree`, and create and add new items at that time. Similarly, when an item is returned at process level, the pool’s item `ret` function can check the item free list count against `maxfree` and trim the item at that time.

However, when the system is busy acquiring items at interrupt level, various situations can influence whether the pool can grow to satisfy the `minfree` value, such as one of the following:

- The item `get` function vector attempts to create new items using memory blocks (`malloc()` and related calls), but the memory manager prohibits obtaining new memory blocks at interrupt level. In this case, the `get` function can queue a request to grow the free list at process level (`pool_request_growth()` internal function), but the free list would still fall below the minimum until the growth task is scheduled.
- The item `get` function vector attempts to create new items using memory chunk elements (`chunk_malloc()` function), and can successfully request a chunk element from which to create a new item. The pool maintains the minimum number of items.
- The item `get` function vector uses chunk elements but there are none available, so it is unable to create a new item at that time. As in the `malloc()` situation, the `get` function can call `pool_request_growth()` to queue a growth request, but in the meantime the free list would fall below the minimum.

The generic pool management code and pool client services attempt to keep a dynamic pool from growing too large and monopolizing memory resources by periodically calling the `pool_prune()` function to trim temporary items from the free list when it exceeds the `maxfree` value.

Trimming operations check the `permtarget` value, set by the `pool_adjust()` `permcount` parameter, to assure that there will always be a certain number of items that are never trimmed.

5.2.9 Obtain an Item from a Pool

Obtain an item from a pool by invoking the item's `get` function vector. See subsection 5.2.7.4, "Pool Item Function Vectors", for more information about a pool's function vector block.

Most pool clients use the packet buffer or particle API's, which define their own item function vectors to get available buffers. If you need to write code for new pool client services, the generic pool management code has the internal functions listed below that can be used in item `get` functions. These are not published as APIs, so consult the pool code in `pool.c` and other pool client service `get` routines (such as `pak_pool_item_get()` in `buffers.c`) for parameter and usage details:

- `pool_dequeue_item()`: Extracts an item from the given pool's available items queue.
- `pool_request_growth()`: Puts in a request to process-level pool service code to replenish the pool with temporary available items, if the pool is dynamic.
- `pool_enqueue_item()`: Adds an item to the pool's available items queue.

For example, the packet buffer item `get` function vector, `pak_pool_item_get()`, calls `pool_request_growth()` to put in a request to try to add more buffers at process level if the free count is too low, and calls `pool_dequeue_item()` if there is an available buffer to retrieve for the caller.

5.2.10 Return an Item to a Pool

Return an item to a pool by invoking the item's `ret` function vector. See subsection 5.2.7.4, "Pool Item Function Vectors", for more information about a pool's function vector block.

Most pool clients use the packet buffer or particle API's, which define their own item function vectors to return available buffers. If you need to write code for new pool client services, see subsection 5.2.9, "Obtain an Item from a Pool", above, for a list of internal pool management functions that can be used in `ret` functions. These are not published as APIs, so consult the pool code in `pool.c` and other pool client service `ret` routines (such as `pak_pool_item_ret()` in `buffers.c`) for parameter and usage details.

5.2.11 Pool Caches: Overview

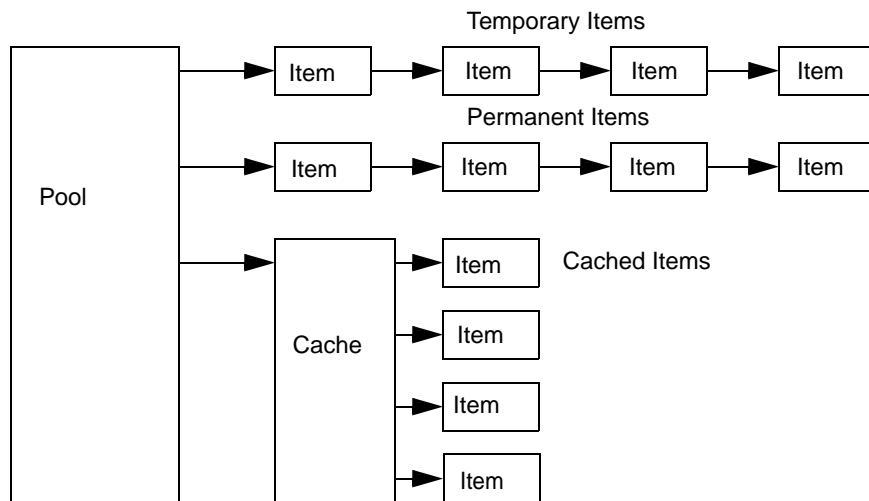
Many of the performance-critical sections of Cisco IOS switching code rely on packet buffer and particle pools for supplying data buffers. Allocating items by calling the pool `get` function vector can impose a considerable overhead in paths where every microsecond counts. To reduce this overhead, the pool management code supports pool item caches. A *pool cache* is a list of free items that can be accessed quickly.

Pool caches were designed for private pools that are typically used in fast packet forwarding applications, so by convention pool caches are used only with private pools. 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.

5.2.12 Structure of a Pool with a Cache

Figure 5-2 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 Figure 5-1. The cache, shown at the bottom of Figure 5-2, is an array of pointers. This structure allows items to be retrieved and returned faster, but requires more careful management by the pool client to avoid resource contention or interrupt timing problems.

Figure 5-2 Structure of Pool with a Cache



5.2.13 Add a Pool Cache

To add a cache to a pool, use the `pool_create_cache()` function. Generally, pool client services such as packet buffer and particle services provide wrapper APIs around this function to initiate buffer cache support, so pool clients usually do not call this function directly. See subsection 5.3.14, “Create and Populate a Packet Buffer Cache”, and subsection 5.4.5, “Create and Fill a Particle Cache”, for information on setting up packet buffer and particle pool caches.

Adding a buffer cache to a pool initializes the structures and internal state required for the pool cache container, but does not add any items to the cache. To fill the cache (or adjust cache parameters later), use the `pool_adjust_cache()` function.

```

boolean pool_create_cache(pooltype *pool, int maxsize,
                         pool_cache_vectors *cache_item,
                         int threshold);

```

Pool cache operations are defined by the function vectors in the `cache_item` parameter, which are listed in Table 5-5. See subsection 5.2.18, “Handle Throttling Conditions in a Pool with a Cache”, for more information about how and when to use the `threshold` parameter along with the `cache_item` threshold vector.

Table 5-5 Pool Cache Item Function Vectors

Vector	Description
get	(Mandatory) Adds an item to the pool cache. This action usually fetches an item from the associated pool's free list for use in the cache.
ret	(Mandatory) Deletes an item from the pool cache. This action usually attempts to return the item to the associated pool's free list.
threshold	(Optional) Provides flow control management. This action is called when the pool cache rises above its optional threshold set at creation. See subsection 5.2.18, "Handle Throttling Conditions in a Pool with a Cache", for more information about situations when the threshold callback action might apply.

The `pool_cache_vectors` function prototypes for pool actions are as follows:

```
typedef void * (*pool_cache_get_t)(pooltype *pool);
typedef void   (*pool_cache_ret_t)(void *item);
typedef void   (*pool_cache_threshold_t)(void);

typedef struct pool_cache_vectors_ {
    pool_cache_get_t      get;
    pool_cache_ret_t      ret;
    pool_cache_threshold_t threshold;
} pool_cache_vectors;
```

5.2.14 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 can also be used to change the size of the pool cache.

```
void pool_adjust_cache(pooltype *pool, int newsize);
```

Items that are placed into the cache list are not available to any of the `get` or `ret` vector actions on the pool. Some drivers use this limitation to implicitly protect packet buffer or particle pool resources for handling incoming traffic, preventing others from accessing the cached items.

Pool client services that support caches can automatically attempt to fill a cache with items that are returned to the pool with which the cache is associated. For example, when the packet buffer `retbuffer()` function is returning a packet buffer to the pool, it checks to see if the pool has a cache that is not full, and if so adds the returned item to the cache rather than to the regular free list.

The `pool_cache_available_items()` function returns the index in the cache array of the last available item in the cache at a particular time.

The `pool_grow_cache()` and `pool_shrink_cache()` internal functions call the pool's cache item `get` and `ret` function vectors to add items to or remove items from the cache.

5.2.15 Obtain an Item from a Pool Cache

Call the `pool_dequeue_cache()` function to obtain an item from the cache associated with the pool specified by the `pool` parameter.

```
void *pool_dequeue_cache (pooltype *pool);
```

This static inline function quickly returns a pointer to the last available item in the cache array.

5.2.16 Return an Item to a Pool Cache

Call the `pool_enqueue_cache()` function to return the pool item specified by the `item` parameter to the cache associated with pool.

```
boolean pool_enqueue_cache (pooltype *pool, void *item);
```

This static inline function quickly sets the next available element in the cache array to point to the returned item.

5.2.17 Destroy a Cache

A cache cannot be destroyed alone. You can only destroy a pool cache if you destroy the pool associated with it by calling the `pool_destroy()` function:

```
void pool_destroy(pooltype *pool);
```

However, you can effectively close down a cache without destroying the pool by calling the `pool_adjust_cache()` function with the `newsize` parameter set to 0.

5.2.18 Handle Throttling Conditions in a Pool with a Cache

At times, a network device might become overloaded, and a pool has no available items, whether the pool is dynamic or static. Often device drivers will handle this condition by *throttling* or shutting down the receive side of the interface. When an interface is throttled, the hardware interface descriptor block (hwidb) is placed on a throttled hwidb queue. A task watches that queue (`net_background`), and is scheduled to dequeue HWIDBs and re-enable them, so packets can again be received on that interface. It can take a while for a throttled interface to be re-enabled because the throttled queue handler runs at normal priority, `PRIO_NORMAL`, and will not be scheduled until all other higher priority tasks (`PRIO_CRITICAL` and `PRIO_HIGH` priority) have been serviced. By the time the interface is re-enabled, conditions that prompted the throttling are likely to have long since been satisfied.

To get an interface with a buffer pool cache back into operation sooner, set a cache threshold value and threshold callback action to facilitate a throttle release in either of the following ways:

- When initializing the pool cache for an interface, call `pool_create_cache()` with a `threshold` value and a threshold vector in the `pool_cache_vectors` vector block. See subsection 5.2.13, “Add a Pool Cache”, for details on using this function to add a cache to a private pool.
- After a pool cache has already been set up, call the `pool_set_cache_threshold()` API function to specify the threshold callback action and cache threshold value to associate with the pool:

```
boolean pool_set_cache_threshold(pooltype *pool, int threshold,
                                pool_cache_threshold_t item_threshold_routine);
```

This function sets *threshold* as the number of available items in the cache at which to invoke callback action *item_threshold_routine*.

If a pool cache threshold value and vector have been set for a pool with a cache, the pool cache manager checks the threshold when returning an item to the cache, and invokes the callback action when the threshold is reached.

The callback action re-enables the interface by calling the function vector `hwidb->enable`, or setting the interface registers. It should be defined as follows:

```
void item_threshold_routine(pooltype *pool)
```

In general, out-of-buffers throttling and re-enabling using the pool cache threshold operates as follows:

- 1 The network interface driver initialization code sets up a private pool, and establishes the threshold value and callback action.
- 2 During system operation, a high level of traffic depletes the interface's private pool. Many items from this pool are in use somewhere else in the system, such as by a forwarding application, in the queuing methodology on various outbound interfaces, or waiting on a transmit (*Tx*) ring.

The driver detects this condition and "throttles" the interface, disabling it from moving packets through the system (effectively, dropping them). If supported by the interface, the transmit component can continue to operate, and forwarding can continue for packets from other interfaces, increasing overall system availability.

- 3 As items are returned to the private pool, the pool manager checks the number of available items in the cache against the threshold, and invokes the threshold callback action if indicated.
- 4 The threshold callback action re-enables interface receive handling and the interface resumes normal operational status.

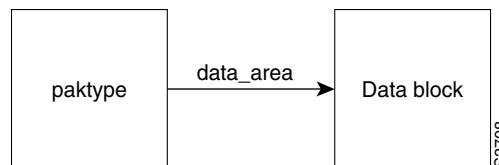
5.3 Packet Buffer Management

The main application of Cisco IOS pools is to manage switching and forwarding of packets. To optimize management of packets, the Packet Buffer Services use a packet buffer structure that has two parts:

- Packet Buffer Headers: Contain IOS-specific information to manage a packet
- Packet Buffer Data Blocks: Contain the packets

Figure 5-3 illustrates the packet structure.

Figure 5-3 Packet Structure



5.3.1 Packet Buffer Headers

A packet buffer header, described by the `paktype` structure, contains the context of the associated packet data and pointers to mark encapsulation boundaries in the data. The header usually resides in main system memory and is managed as part of the `MEMPOOL_CLASS_LOCAL` memory pool.

The `paktype` structure is highly used, and its fields are referenced often during interrupt-level forwarding operations. As a result, the locality of certain structure members is carefully tuned to achieve maximum performance during forwarding. The `PAKBASE_DEF` macro in `pakbase.h` is provided to identify and optimize the locality of the performance-critical members of the `paktype` structure that are heavily used by interrupt-level forwarding and switching code.

The remaining members of the `paktype` structure represent other context for the packet. Section 5.3.2, “Packet Buffer Data Blocks”, describes some of the most commonly used fields.

Particle-based packets are also associated with a `paktype` header to maintain the context of the whole packet, although they do not reference a packet data buffer. See subsection 5.4.2, “Particle Structure”, for details.

5.3.2 Packet Buffer Data Blocks

A Cisco IOS buffer data block contains the actual packet itself and is referred to as a packet data block, or *pakDB*. The data is an untyped block of memory that can reside in either main memory or a shared area of memory usually referred to as *IOMEM*. The choice of memory used for packet data blocks is platform-dependent.

All references to the data block are made using the `paktype` header, because its fields map the protocol-specific contents of the buffer. The `data_area` member in the `paktype` structure points to the base of the packet data block. The packet data block can be deciphered and manipulated using other `paktype` fields depending on the protocols being used.

5.3.2.1 Organization Within a Data Block

Within a data block, the packet is organized as shown in Figure 5-4. This figure shows limited data buffer state information; the `paktype` header maintains a much more extensive context of the actual data buffer.

The three shaded areas in the figure indicate a single network packet in the buffer. Packets are represented in the `data_area` member of a `pakdata` structure, and contain three consecutive memory areas:

- An encapsulation (for example, an Ethernet ARPA header)
- A network header (for example, an IP header)
- A payload (for example, a UDP datagram)

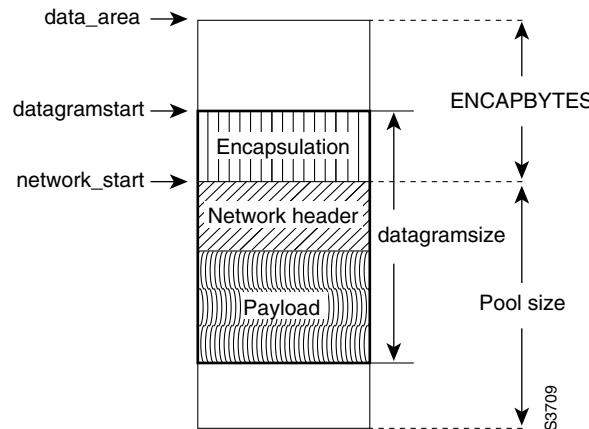
The `data_area` member of the `paktype` header points to the first usable byte in the data block. It is preceded by a *magic number* that is used to detect packet corruption due to erroneous overflow assignments from other memory blocks in the system.

The `datagramstart` member points to the beginning of the packet itself, which is the protocol-dependent frame header encapsulating the network data. The total size of the frame in memory is stored in the `datagramsize` member.

The `network_start` member contains a pointer to the start of the network header in the packet.

Other members of the `paktype` header mark additional significant locations in the data that can be determined, such as the beginning of upper-layer protocol headers within the packet (for example, `transport_start` or `session_start`).

Figure 5-4 Memory Organization within a Data Block



As shown in Figure 5-4, the packet does not start at the first available byte in the data block, where `data_area` points. Rather, the actual start of the packet is offset to allow for expansion of the frame header, for example, to allow insertion of additional information for tunneling, or to change the encapsulation to a larger frame header. The offset, defined by the `ENCAPBYTES` constant, allows protocol and interface code to use the extra encapsulation space at the beginning of the data block to write larger headers without the expense of copying the entire packet.

In addition to encapsulation expansion space at the beginning of the packet, some interfaces require space after the last data byte for fields such as checksum data, and padding out to an alignment boundary. The extra trailing space is defined by the `TRAILBYTES` constant.

To effectively handle these additional memory requirements, the Packet Buffer Services automatically increase the requested size of the data buffer with the following definition:

```
#define BUFFEROVERHEADBYTES (sizeof(pakdata) + ENCAPBYTES + TRAILBYTES)
```

The requested size of the data buffer is specified as a parameter to the packet buffer pool creation function, `pak_pool_create()`, and is usually only the maximum of the network header plus the associated payload. The Packet Buffer Services add the additional bytes for the “buffer overhead”, which is the expanded encapsulation space and trailing space.

When a packet buffer is created, the following fields in the `paktype` structure are initialized for the new packet buffer (shown as a variable packet buffer pointer called `pak`):

```
pak->datagramstart = pak->data_area + ENCAPBYTES  
pak->network_start = pak->data_area + ENCAPBYTES
```

When possible, packet buffer client code attempts to place the beginning of the network header of the datagram (referenced by `network_start`) at an offset of `ENCAPBYTES` bytes from the start of the data area. At this location, the packet is considered to be “centered”. However, this location cannot be guaranteed for various reasons, so packet consumers should always use `network_start` and other `paktype` pointers to locate specific boundaries for data in a buffer. Valid packet data can be located started at `datagramstart` and ending at `datagramstart + datagramsize`.

5.3.3 Packet Buffer Pools: Overview

The Cisco IOS Packet Buffer Services use generic pool management operations to manage contiguous buffers for receiving and transmitting data. Each packet buffer pool holds packet buffers that are the same size and have the same properties.

At system initialization, the Packet Buffer Services sets up 6 public packet buffer pools for general use by any system component. These public pools can be destroyed or modified by the platform initialization code. Later during system initialization, additional packet buffer pools might be created, depending on the platform and system component requirements.

After initialization, the Packet Buffer Services maintain the shared buffer pools and any additional public or private pools. The Packet Buffer Services API includes functions that allow you to do the following:

- Get packet buffers from the preallocated shared buffer pools and return them.
- Create and use additional shared (*public*) packet buffer pools for optimizing packet data processing.
- Create and use exclusive (*private*) packet buffer pools, with or without a pool cache, for optimizing performance and buffer availability for network interfaces.
- Set up a fallback pool from which to get packet buffers if the preferred pool has no free buffers.
- Lock a packet buffer.
- Copy a packet buffer.
- Associate a packet buffer with an interface.

5.3.3.1 Public and Private Packet Buffer Pools

Packet buffer pools are classified as either *public* or *private*.

Public packet buffer pools allow any packet buffer pool client to allocate buffers from their pools, and are identified by the public pool group number `POOL_GROUP_PUBLIC`, which is 0.

Private packet buffer pools have unique identifiers, and are accessible only by packet buffer pool clients that know about them. These pools are used primarily by interface drivers to manage buffers of specific MTU sizes for handling incoming and outgoing traffic.

5.3.4 Initialize System-wide Public Packet Buffer Pools

At system initialization, the `pak_pool_init()` function allocates a set of 6 public packet buffer pools for general use that will accommodate common MTU sizes for that platform, with default buffer sizes defined by constant values `SMALLDATA` through `HUGEDATA`. The packet buffer headers are allocated from system memory and the packet data blocks from `IOMEM`, if available.

The `pak_pool_init()` function calls the `platform_buffer_init()` function for setting up pools according to platform-specific parameters. The platform-specific initialization code might destroy and reconfigure some or all of the shared general packet buffer pools, and might set up additional private pools for its interfaces.

When you are developing code for a new platform or interface, tailor `platform_buffer_init()` to create an optimal buffer management system on your platform. Network interface code commonly uses a private pool with a packet buffer cache, and falls back to allocating packet buffers from preallocated public pools if its private pool is exhausted.

After initialization, the Packet Buffer Services maintain the system-wide public packet buffer pools and any other packet buffer pools and caches initialized through packet buffer API calls.

5.3.5 Create a Public Packet Buffer Pool

You might want to create your own public pools of packet buffers to optimize packet forwarding and switching for a particular environment. If you create new pools, you can also use the Packet Buffer Services public pools at any time for normal or fallback buffer allocation, depending on the MTU size you need for a particular buffer request.

To create a new public buffer pool for packet buffers, follow these steps:

Step 1 Create a new packet buffer pool by calling [pak_pool_create\(\)](#).

```
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, and supplies standard function vectors to create, destroy, get, and return buffers for packet buffer pool management.

To create a public buffer pool, specify a `group` parameter of `POOL_GROUP_PUBLIC`.

Although they all have the same group number, public packet buffer pools are distinguished from each other by their buffer size. The `size` parameter specifies the data size of buffers contained in the pool. See subsection 5.3.2.1, “Organization Within a Data Block”, for more information on defining the buffer size. If the size is 0, only buffer headers are created in the pool.

Specify `NULL` for the `mempool` parameter to allocate buffer data blocks from IOMEM when populating the pool.

See the [pak_pool_create\(\)](#) page in the *Cisco IOS API Reference Guide* and Table 5-2 for details on supported values for the `flags` parameter.

The `pak_pool_create()` function creates the pool management `pooltype` structure, but does not populate the pool with available packet buffers. It associates the internal packet buffer functions to create, destroy, get, and return packet buffers with the newly created pool, so the generic pool management code invokes them when API functions such as `pool_adjust()`, `getbuffer()`, and `pool_getbuffer()` are called.

Step 2 Populate the pool with packet buffers by calling `pool_adjust()`.

Populating the pool involves a pair of memory allocation functions, from local system memory for the `paktype` header structure and the memory region class defined in `pak_pool_create()` for the correct size of the packet data buffer.

During an interrupt, the generic pool manager cannot allocate new memory blocks. To avoid running out of packet buffers while executing at interrupt level, carefully set the minimum number of available buffers (`mincount` parameter) when calling `pool_adjust()` so that the buffer pool manager always has enough buffers to meet the demands of the interrupt code. See subsection 5.2.8.1, “Pool Parameters in Dynamic Pools”, and section 5.5, “Useful Buffer Commands”, for more information about judiciously setting and checking packet buffer pool parameters to assure optimal buffer management.

5.3.5.1 Example: Create a Public Packet Buffer Pool

The following example creates a public pool called “Large” and contains buffers of size of LARGEDATA bytes. This means that the size of the data area available for writing network frames is `sizeof(pakdata) + (ENCAPBYTES + LARGEDATA + TRAILBYTES)` because the Packet Buffer Services account for the packet data area magic number and extra encapsulation and trailing padding. However, refer to the pool’s buffer size as the MTU size (LARGEDATA in this case). This pool is set up with default flags, which specify pool type dynamic and to allocate the header and data blocks from memory pools. Specifying `mempool` as `NULL` tells the pool create function to allocate buffer data blocks from the packet buffer default memory pool, which is `MEMPOOL_CLASS_IOMEM`.

```
large = pak_pool_create("Large", POOL_GROUP_PUBLIC, LARGEDATA,  
                        POOL_DEFAULT_FLAGS, NULL);
```

5.3.6 Create a Private Packet Buffer Pool

You might want to create private packet buffer pools to optimize buffer management for forwarding and switching. Other interfaces that do not have access to the private pool are prevented from using those buffer resources. If you create new private pools, you can also still use the Packet Buffer Services public pools at any time for normal or fallback buffer allocation.

Creating a private buffer pool first requires obtaining a private pool group ID to pass to the pool creation function.

To create a private buffer pool, follow these steps:

Step 1 Obtain a new private pool group number by calling `pool_create_group()`.

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 assigned by the `pool_create_group()` function.

Step 2 Create a new pool by calling `pak_pool_create()` with the private group ID.

`pak_pool_create()` sets up the pool parameters and associates the operation vectors with the pool in the same way as when creating a public packet buffer pool, storing the group ID in the `pooltype` structure. It does not populate the pool with available packet buffers.

Step 3 Populate the private pool with packet buffers by calling `pool_adjust()`.

Populating the pool involves a pair of memory allocation functions, from local system memory for the `paktype` header structure and the memory region class defined in `pak_pool_create()` for the correct size of the packet data buffer.

During an interrupt, the pool manager cannot allocate new memory blocks. To avoid running out of packet buffers while executing at interrupt level, carefully set the minimum number of free buffers when calling `pool_adjust()` so that the buffer pool manager always has enough buffers to meet the demands of the interrupt code. See subsection 5.2.8.1, “Pool Parameters in Dynamic Pools”, and section 5.5, “Useful Buffer Commands”, for more information about judiciously setting and checking packet buffer pool parameters to assure optimal buffer management.

Step 4 Often a cache is associated with a private packet buffer pool to optimize buffer access. See subsection 5.3.14, “Create and Populate a Packet Buffer Cache”, for details on setting up and using a cache with a packet buffer pool.

- Step 5** Private packet buffer pools are often associated with a *fallback pool*, which is a public packet buffer pool with buffers of the right size from which to retrieve a buffer if the attempt to get one from the private packet buffer pool fails. See subsection 5.3.6.1, “Find and Assign a Fallback Packet Buffer Pool”, for details on setting up a fallback pool.

5.3.6.1 Find and Assign a Fallback Packet Buffer Pool

To find the public packet buffer pool that is the best fit for a given packet buffer size, use the `pak_pool_find_by_size()` function.

```
pooltype *pak_pool_find_by_size(int size);
```

The main use of this function is to find and set up a fallback public pool from which to retrieve packet buffers if an attempt to retrieve one from an optimally-sized private packet buffer pool fails. To find an appropriate public buffer pool, pass the maximum expected packet size to the `pak_pool_find_by_size()` function, and store the returned `pooltype` pointer in the private pool structure’s `fallback` member, as shown in the following example:

```
private_pool->fallback = pak_pool_find_by_size(max_packet_size);
```

5.3.6.2 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 last parameter is `NULL`, indicating that the data area for each buffer is located in the default memory class pool for packet buffers, `MEMPOOL_CLASS_IOMEM`.

```
pool_group = pool_create_group();
buffer_pool = pak_pool_create("Ethernet", pool_group, MAXETHERSIZE,
POOL_SANITY, NULL);
```

5.3.7 Obtain a Packet Buffer

When a private packet buffer pool exists and is associated with a cache, this is usually the first choice of location from which a packet buffer client attempts to obtain a packet buffer. If there is no cache, the next choice is from the private packet buffer pool directly, and the last choice is from a public packet buffer pool with a size closest to the required size.

Use the guidelines in Table 5-6 when implementing packet buffer pool client code to obtain a packet buffer, referring to the indicated subsections for more details.

Table 5-6 Guidelines for Obtaining Packet Buffers

API Function to Invoke	Purpose	See for Details
<code>pool_dequeue_cache()</code>	First, if a cache is present for a private buffer pool, try this function. Cached buffer retrievals result in the fastest acquisition and optimal buffer size by design.	Section 5.3.15, “Access Packet Buffers in a Cache”
<code>pool_getbuffer()</code>	Next, if the cache retrieval fails, or there is a private packet buffer pool without a cache, use this function to get a buffer directly from the private pool. Private pools are designed to have the optimal buffer size for a particular platform or code task.	Section 5.3.7.2, “Obtain a Packet Buffer from a Private Buffer Pool”

Table 5-6 Guidelines for Obtaining Packet Buffers (continued)

API Function to Invoke	Purpose	See for Details
getbuffer()	Lastly, if the private packet buffer retrieval fails or there is no private packet buffer pool, use this function to obtain a packet buffer from one of the public packet buffer pools. This function finds the first pool with a large enough buffer size that has an available item.	Section 5.3.7.1, “Obtain a Packet Buffer from a Public Buffer Pool”
pak_pool_find_by_size()	Use this function to find a fallback public packet buffer pool, from which to retrieve an available buffer of as close as possible to the right size from among all public buffer pools in the system, when an attempt to retrieve a buffer from a private pool fails.	Section 5.3.6.1, “Find and Assign a Fallback Packet Buffer Pool”
pak_lock()	If several networking applications are going to use a particular packet buffer, or an application needs to keep a packet buffer around, use this function to increment the reference count. A packet buffer is not returned to its pool until the reference count is decremented to zero.	Section 5.3.8, “Lock and Unlock a Packet Buffer”

The following example shows a typical packet buffer client attempting to retrieve a packet buffer first from a private pool cache, then if that fails, tries the private pool directly, and finally, if that fails, tries the system’s public packet buffer pools.

```
if ((pak = pool_dequeue_cache(my_pool)) == NULL) {
    pak = pool_getbuffer(my_pool);
    if (!pak) {
        pak = getbuffer(sizeof(mybuffer));
    }
}
```

5.3.7.1 Obtain a Packet Buffer from a Public Buffer Pool

The simplest packet buffer requests in the system image use the public packet buffer pools. To obtain a packet buffer from among the public buffer pools, use the `getbuffer()` function.

```
paktpe *getbuffer(int size);
```

The `size` parameter is the amount of space required for the network header and payload only; each buffer from the default Packet Buffer Services comes with enough space for the largest expected encapsulation + TRAILBYTES + `sizeof(pakdata)`. The `getbuffer()` function scans all the public buffer pools in an attempt to find a pool with data buffer size of at least `size` bytes. If it finds a matching pool that has fewer than the minimum number of free buffers, and the pool is dynamic, it attempts to increase the pool to the minimum free count before returning a buffer.

The `paktpe` structure is returned by `getbuffer()` with applicable data pointers already initialized, such as the `network_start` and `datagramstart` fields, taking into consideration the expected encapsulation space. The pointers are ready to be used to build packets for transmission.

Note Always check the return value from `getbuffer()`. Even with dynamic pools, it is possible for buffer resources to be exhausted and `getbuffer()` to fail due to low memory or heavy burst conditions in a running system.

The following example allocates a packet 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. After 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 is a typical code example of building buffers using `getbuffer()` or related functions. Note that `getbuffer()` does not zero the data area of the buffer, and all parts of a network header and its payload must be written for each frame.

5.3.7.2 Obtain a Packet Buffer from a Private Buffer Pool

To obtain a packet buffer from a private pool, call the `pool_getbuffer()` function, passing a pointer to the private pool from which to obtain the buffer.

```
paktype *pool_getbuffer(pooltype *pool);
```

If it cannot retrieve a packet buffer from the pool specified by the `pool` parameter, and a fallback pool is assigned in `pool->fallback`, `pool_getbuffer()` attempts to find a buffer from the fallback pool. Subsection 5.3.6.1, “Find and Assign a Fallback Packet Buffer Pool”, describes how to set up a fallback pool.

5.3.8 Lock and Unlock a Packet Buffer

To increment the reference count field (`refcount`) of a packet buffer, which indicates that it is still in use and should not be completely released until the reference count is 1, use the `pak_lock()` macro.

```
pak_lock(pak);
```

To free the lock, call `datagram_done()` for this packet buffer. See subsection 5.3.9, “Return a Packet Buffer to a Pool”, for details and special considerations for locking and releasing packet buffers.

5.3.9 Return a Packet Buffer to a Pool

In most situations, to return a packet buffer to a pool, use the `datagram_done()` function.

```
void datagram_done(paktype *pak);
```

This function applies to public buffer pools, private buffer pools, and caches, and is considered a secure way to return a packet buffer. It acts upon the value of the reference count in the `paktype` header, `refcount`, as follows:

- `refcount > 1`: The reference count is decremented and the packet is not returned to the originating buffer pool. In this case, `refcount` indicates how many system components are still using the packet buffer.

- `refcount = 1`: This indicates no other system components are still using this packet buffer, and it can be returned to the originating buffer pool.

When a packet buffer is returned, it is always returned to its originating buffer pool via a pointer in the `paktype` header structure, `paktype->pool`, with the following actions:

- If returning a packet buffer to a private pool that has a private buffer cache that is not fully populated, the packet buffer is placed in the cache array.
- If returning a packet buffer to a private pool with a cache and the cache is full, or there is no cache, the packet buffer is returned to the original private pool itself.
- A public packet buffer is returned directly to the pool from which it was obtained.

Another packet buffer release function, `retbuffer()`, is available, but should only be used with caution. This function essentially expects that the `refcount` for the packet buffer is always 1, in other words, that only one component references this packet buffer at a time. Because interrupt-level forwarding is a compact path, often you might be fairly certain that there are no other packet buffer clients using a packet buffer or that are still processing it and need to retain it beyond the execution path you are coding. The `retbuffer()` function might be used in that case.

```
void retbuffer(paktype *pak);
```

This function acts differently depending on the value of the reference count, but unlike `datagram_done()`, it does not decrement `refcount`, as follows:

- `refcount != 1`: An error message is generated, and `retbuffer()` does not release the packet.
- `refcount = 1`: The packet buffer is returned to the originating pool in the same way as described above for `datagram_done()`.

Other packet buffer clients that want to process a packet buffer that they did not obtain themselves can prevent it from being released by calling `pak_lock()`, which increments `refcount`. However, if the clients accessing the buffer use `retbuffer()` to release buffers when they are done with them, packet buffer leaks (translates into memory leaks) can occur because that function does not decrement `refcount` nor release packet buffers if `refcount > 1`.

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()` is the reliable packet buffer unlock function that is used by packet buffer clients to relinquish their hold on a buffer, with the last one actually returning the buffer.

5.3.9.1 More Guidelines for Returning a Packet Buffer

Follow these guidelines when designing code to return a packet buffer:

- When returning a packet buffer, the best practice is to use `datagram_done()`.
- Do *not* make decisions based on the reference count, and do *not* add an explicit unlock. If a packet buffer is being shared, you cannot make assumptions about what the other packet buffer clients are doing or what order they will be done with the packet. You should not attempt to retrieve or make code decisions based on the reference count.
- Do *not* add any explicit unlocks, to avoid risk of race conditions. Unlocking and releasing must be integrated.

5.3.10 Reuse a Packet Buffer

The `pak_reset()` and `pak_common_cleanup()` functions are used to clear commonly-used packet buffer header fields in order to reuse a packet buffer when all processing on it is complete. The Packet Buffer and Particle Services call `pak_reset()` internally when a packet buffer is created or returned to a packet buffer pool.

These functions can also be used when a platform manages its own packet buffers (for example, the C7500 RSP). The `pak_common_cleanup()` function is a lightweight (less CPU-intensive) version of `pak_reset()` to use in interrupt-level packet handling. The `pak_subblock_set()` and `pak_subblock_reset()` API functions help efficiently manage packet subblock cleanup in `pak_common_cleanup()`.

Refer to the API function descriptions for complete usage information for calling these functions.

5.3.11 Prune a Packet Buffer Pool

Over time, the pool manager works with the Packet Buffer Services to try to reclaim unused packet buffers from dynamic pools that contain more free packet buffers than their defined `minfree` count. Periodically, the internal `pak_pool_periodic()` function runs at process level to call the `pool_prune()` function for any public and private packet buffer pools in the system. The `pool_prune()` function examines the free items in a pool and return any that haven't been used during the previous interval defined by `POOL_PRUNE_TIME` (a platform-independent constant currently set to 80 minutes defined in `pool.h`). See subsection 5.2.8.1, "Pool Parameters in Dynamic Pools", for more information.

All currently allocated buffers constitute memory that is "in use" (as seen in `show mem` output), even when those buffers are on their pool's free list. If the system is short of memory, more aggressive pruning of free buffers might be appropriate.

5.3.12 Duplicate or Expand a Packet Buffer

Often you need to duplicate a packet buffer so that it can be sent to multiple destinations or modified without the risk of modifying a buffer that might be waiting in a transmit queue. When you need to guarantee that a frame's contents are valid when it is being multiply referenced, you should copy the frame. It can take tens of milliseconds to transmit some frames after sending them to a driver, especially on slow serial links. As a result, no further manipulation on a buffer can take place after that buffer is handed to a driver. The only way to asynchronously manipulate the same basic buffer contents is to copy the buffer repeatedly before sending the frame, and then continue working on the duplicate.

Packet buffer duplication functions might be used in the following situations:

- *Packet fragmentation and reassembly*: Often required when a packet is switched from an interface using a large MTU size to an interface using a smaller MTU size.
- *Broadcasting and multicasting*: When a packet must be duplicated so that it can be transmitted through multiple interfaces.
- *Tunneling*: Packets that have been received on one interface can be tunneled and transmitted out another interface. If the input interface has a larger MTU size than the transmitting interface, the packet might need to be fragmented.

The Packet Buffer Services provide a variety of functions for duplicating buffers that include different combinations of the following operations:

- Copy a packet buffer header.

- Copy the data area of a packet buffer.
- Realign or *recenter* the data area of a packet buffer when duplicating it.
- Reassign the buffer header pointers in a duplicated buffer.
- Increase the size of a packet buffer.

Table 5-7 summarizes the characteristics of the packet buffer API functions related to duplicating or copying packets.

Table 5-7 Summary of Packet Buffer Duplication Functions

Packet Buffer Duplication Function	Size of Data Copied	Allocates Destination Buffer	Recenters Packet	Returns Source Buffer to Pool
<code>pak_copy()</code> Copies the requested size of the data area to the given destination packet buffer.	<code>size + ENCAPBYTES + network_start offset</code>	No	No	No
<code>pak_copy_and_recenter()</code> Copies the requested size of the data area to the given destination packet buffer, recentering or realigning <code>network_start</code> to the <code>ENCAPBYTES</code> offset within the data area.	<code>size + ENCAPBYTES + network_start offset</code>	No	Yes	No
<code>pak_duplicate()</code> Allocates a new destination packet buffer of the same size as the source buffer's pool size, and copies the entire data area, without recentering the frame.	<code>pool size + ENCAPBYTES + network_start offset</code>	Yes	No	No
<code>pak_clip()</code> Allocates a new destination packet buffer of the requested size from a public buffer pool, and copies that size of the data area, without recentering the frame. Assures the destination always has enough space for the requested size.	<code>copy_size + ENCAPBYTES + network_start offset</code>	Yes	No	Mp
<code>pak_grow()</code> Returns the source buffer if it is large enough for the new size, otherwise gets a destination buffer of the right size from a public packet buffer pool, copies the requested size to the destination, releases the source buffer (calls <code>datagram_done()</code>), and returns the new destination buffer.	<code>oldsize + ENCAPBYTES + network_start offset</code>	If source buffer is not large enough	No	Yes, if new destination buffer was allocated

When copying a source packet buffer to a new destination packet buffer, these functions save certain fields in the duplicate buffer's packet header that are not related to the alignment or contents of the data area, such as the pool pointers and flags, then copy the entire packet header in a block from the source buffer to the new buffer, and restore the saved fields to the destination buffer header. After copying the data area, these functions recalculate and set the destination packet header pointer values to reference the new data area, accounting for any realignment.

These functions add the encapsulation padding (in case this space contains additional encapsulation headers) and the `network_start` offset if the packet data is no longer "centered" (`network_start` no longer aligns with `ENCAPBYTES` from the start of the data area) when determining the actual number of bytes or size of the data area to be copied to the destination buffer.

Note Packet buffer duplication is complicated, so only use Packet Buffer Services API functions to copy a buffer. Do not use `bcopy()` or equivalent functions, which can result in memory corruption in this environment.

Packet buffer duplication functions can fail, so always check return values.

More details on these functions is provided in the following subsections. Also see [Chapter 5, "Pools, Buffers, and Particles"](#) in the *Cisco IOS API Reference*, or the IOS packet buffer code base (`buffers.c`) for more information about these and other available functions related to duplicating packets.

5.3.12.1 Duplicate a Packet Buffer to a Given Destination Buffer

Use the `pak_copy()` function when the duplicate of a packet buffer is not likely to grow. This function copies `datagramsize` bytes of the frame (the network header and payload) plus the encapsulation padding area of the buffer. The `pak_duplicate()` function always copies the entire data area from the source buffer to the duplicate, which can be inefficient if the contents of the original buffer are small relative to the size of the data area. Because `datagramsize` might include some of the encapsulation area, `pak_copy()` might copy a little more data than is required, but can still be more efficient than `pak_duplicate()`.

```
void pak_copy(paktype *source, paktype *destination, int size);
```

This function does not recenter the data area in the duplicate, so if the `network_start` offset in the source buffer is significant and the destination buffer is allocated based on the original payload size, the data might be truncated because it is being copied to a destination buffer that is not long enough. The destination buffer provided to `pak_copy()` must always be large enough to hold the frame size plus the offset of the source buffer's `network_start` from the ideal center of the buffer (the encapsulation offset).

In environments where packet buffers often end up with a larger offset from the significant offsets from the ideal center, you might have better results using `pak_copy_and_recenter()`, or `pak_duplicate()` or `pak_clip()`, which both always obtain a destination packet buffer of the right size for the copy.

Example: Duplicate a Packet Buffer to a Given Destination Buffer

The following example shows how to duplicate a packet buffer to a given destination buffer, with the `pak` pointing to the source buffer, and `newpak` provided for the destination buffer.

```
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;

```

5.3.12.2 Duplicate and Recenter a Packet Buffer

The `network_start` member in the packet buffer header points to the start of the network header of the frame in a packet buffer. The frame is considered to be *centered* when `network_start`, the pointer to the beginning of the frame, is `ENCAPBYTES` from the start of the data area. However, a frame is not always centered, especially if it has been manipulated with different encapsulations. If the buffer contents must be expanded towards the end of the data area, the buffer might run out of space.

To avoid this problem, use the `pak_copy_and_recenter()` function to duplicate a given size of the data area of a packet buffer to a given destination buffer while recentering the frame in the new buffer. This function calls `pak_center()` to realign `network_start` of the destination buffer back to the top of the encapsulation padding boundary when copying the data, effectively snapping the buffer contents back into an optimal position.

```
void pak_copy_and_recenter(paktype *source, paktype *destination, int size);
```

Example: Duplicate and Recenter a Packet Buffer

The following example shows a call to `pak_copy_and_recenter()` that duplicates the packet buffer pointed to by `pak` into the given destination buffer pointed to by `newpak`, recenters the destination frame when copying the data area, then releases the original packet buffer.

```

paktype *newpak;

newpak = getbuffer(pak->datagramsize);
if (newpak)
    pak_copy_and_recenter(pak, newpak, pak->datagramsize);
datagram_done(pak);

```

5.3.12.3 Locate the Ideal Offset or Center Within a Packet Buffer

Use the `pak_center()` function to locate the ideal offset or center of the data area of a packet buffer, which is at an offset of `ENCAPBYTES` from the beginning of the data area.

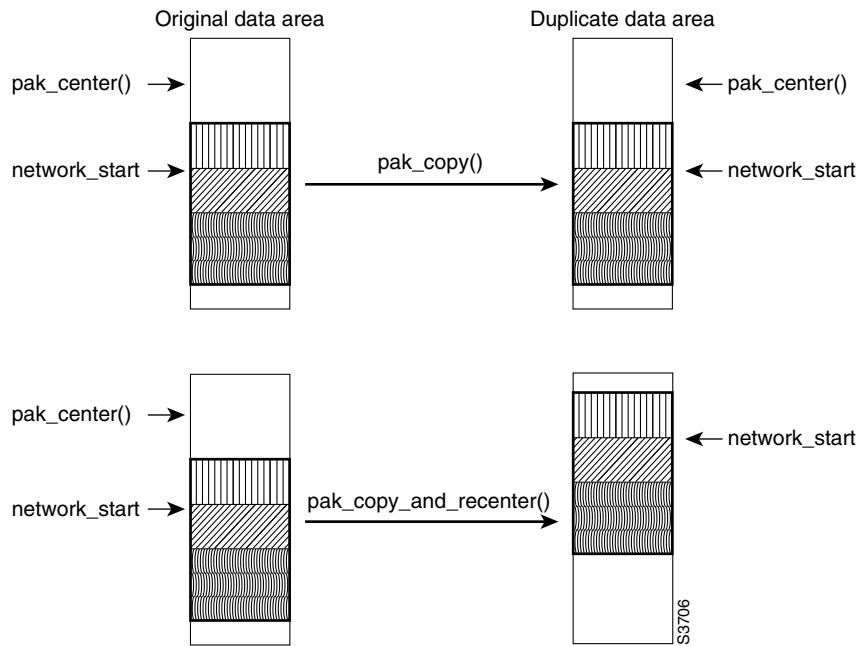
Example: Locate the Ideal Offset or Center Within a Packet Buffer

The following example sets `network_start` of a newly obtained packet to the ideal center of the data area.

```
pak->network_start = pak_center(pak);
```

5.3.12.4 Comparison of Packet Buffer Duplication with and without Recentering

Figure 5-5 compares the effects of the `pak_copy()` and `pak_copy_and_recenter()` functions on the destination buffer. The figure shows the location of the ideal center returned by the `pak_center()` function.

Figure 5-5 Comparing the pak_copy() and pak_copy_and_recenter() Functions

5.3.12.5 Duplicate a Packet Buffer into a New Destination Buffer

The `pak_duplicate()` function obtains a new packet buffer of the same size as the source buffer, and duplicates the source packet buffer into the new packet buffer without recentering the frame in the destination buffer. This function copies all the data area from the source buffer to the duplicate, including the encapsulation padding space.

```
paktype *pak_duplicate(paktype *pak);
```

If the source packet buffer header has the `POOL_PUBLIC_DUPLICATE` flag set for the originating pool, then this function attempts to get the new packet buffer from the public packet buffer pool group (using `getbuffer()`). Otherwise, it attempts to get the new packet buffer from the same pool as the source buffer (using `pool_getbuffer()`).

The `pak_clip()` function is another duplication API function that takes a `copy_size` parameter rather than using the size of the source buffer. Like `pak_duplicate()`, this function obtains a new buffer, but copies the requested size plus any offset from the ideal `network_start` location that might be present in the source buffer, effectively guaranteeing that the destination buffer is always large enough to fit the requested size.

```
paktype *pak_clip(paktype *pak, int copy_size)
```

Example: Duplicate a Packet Buffer into a New Destination Buffer

The following example shows a call to `pak_duplicate()` that duplicates the packet buffer pointed to by `pak` and returns a pointer to the new destination packet buffer, which is stored in `newpak`.

```
newpak = pak_duplicate(pak);
if (newpak == NULL)
    return;
newxns = (xnshdrtype *)xnsheadstart(newpak);
newpak->if_output = pak->if_input;
```

```
newxns->cksum = 0;
```

5.3.12.6 Allow the Size of a Packet Buffer to Grow

To allow the size of a packet buffer to increase, use the `pak_grow()` function.

```
paktype *pak_grow(paktype *pak, int oldsize, int newsize);
```

This function checks the maximum size of the packet buffer pointed to by the `pk` parameter and compares this to the given new buffer size. If the source packet buffer has enough space to accommodate the new size, this function returns the source packet buffer pointer itself as the “new” buffer in which the data area can grow. If the source packet buffer is not large enough to accommodate the new size, `pak_grow()` allocates a new packet buffer of the new size from a public buffer pool (using `getbuffer()`), copies the data area to the new buffer (using `pak_copy()`), releases the original packet buffer (pointed to by `pk`), and returns a pointer to the new packet buffer. If any of those operations fail, `pak_grow()` returns NULL and leaves the original `pk` intact.

Always check the return value of the `pak_grow()` function to determine when the source buffer pointer is still valid; if the return value is a non-NUL pointer that is different from the original source buffer (`pk`), then the source buffer was released and `pk` is no longer a valid pointer.

5.3.13 Packet Buffer Caches: Overview

The Cisco IOS Packet Buffer Services provide API functions that allow you to associate a packet buffer pool with a cache for faster access to available packet buffers. Interface code can take advantage of a cache to bypass some of the packet retrieval overhead. By convention, packet buffer pool caches are used only with private pools. The cache is an array of pointers to available packet buffers in that pool, and for platforms that have FAST memory, can be allocated from that faster memory to further optimize packet handling.

However, buffers stored in the cache are not available for retrieval by `getbuffer` or `pool_getbuffer()` operations on that pool. One drawback of using a cache is that it requires writing additional code for managing cache access, including obtaining and returning its packet buffers, and handling when there are no available buffers in the cache.

5.3.14 Create and Populate a Packet Buffer Cache

To add a buffer cache, use the `pak_pool_create_cache()` function. Adding a packet buffer cache to a pool initializes the structures and internal state required only; it does not add any packet buffers to the cache.

```
boolean pak_pool_create_cache(pooltype *pool, int maxsize);
```

Many factors influence the size (`maxsize`) to specify for the cache. Some examples that have been used include the size of the interface’s receive ring, twice the receive ring size, one-half or one times the number of permanent packet buffers in the pool, all of the packet buffers in the pool, $1024 + \#channels/4$, and many others.

After creating a packet buffer cache, use the `pool_adjust_cache()` function to fill the array with packet buffer pointers. You can also call this function later to repopulate or adjust the number of packet buffers the cache holds. See subsection 5.2.14, “Fill a Pool Cache”, for details on using this function.

5.3.14.1 Example: Create and Fill a Packet 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);
}
```

5.3.14.2 Example: Set up a Private Packet Buffer Pool with a Cache

The following code example shows a typical packet buffer pool implementation.

```
/*
 * Initialize private buffer pool if there isn't one.
 * Populate the pool, initialize a private buffer cache,
 * and populate the buffer cache with buffers from the newly
 * created private pool. Also set up the fallback public buffer
 * pool.
 */
if (!idb->pool) {
    if (idb->pool_group == POOL_GROUP_INVALID)
        idb->pool_group = pool_create_group();

    if (!idb->hw_namestring)
        idb_init_names(idb->firstsw, FALSE);

    lance_buffer_pool = pak_pool_create(idb->hw_namestring,
                                         idb->pool_group, idb->maxdgram, POOL_SANITY | POOL_CACHE, NULL);
    if (!lance_buffer_pool) {
        crashdump(0);
    }
    lance_buffer_pool->fallback = pak_pool_find_by_size(idb->maxdgram);
    pool_adjust(lance_buffer_pool, 0, PRIV_NUM_BUF(ds->rx_size),
                PRIV_NUM_BUF(ds->rx_size), TRUE);

    pak_pool_create_cache(lance_buffer_pool, PRIV_NUM_CACHE(ds->rx_size));
    pool_adjust_cache(lance_buffer_pool, PRIV_NUM_CACHE(ds->rx_size));

    idb->pool = lance_buffer_pool;
}
```

5.3.15 Access Packet Buffers in a Cache

Packet buffer pool operations to obtain a packet buffer from a pool, such as `getbuffer()` and `pool_getbuffer()`, do not consider whether or not a cache is associated with the pool. Instead, interface code that uses a cache must explicitly call the pool cache functions to obtain packet buffers from and return them to a packet buffer cache.

Use the `pool_dequeue_cache()` function to explicitly obtain a packet buffer from a packet buffer pool cache (see subsection 5.2.15, “Obtain an Item from a Pool Cache”).

Use the `pool_enqueue_cache()` function to return a packet buffer to the cache (see 5.2.16, “Return an Item to a Pool Cache”).

The following example shows an example of an inline packet pool client routine that attempts to obtain a packet buffer from a pool with a cache by first checking the cache, and if that fails, tries to obtain it from its private pool free list. This allows the packet buffer cache policy to be set on a per-client basis.

```
static inline paktype *mygetbuf_inline (hwidbtype *myidb, ulong data_offset,
                                         boolean cache_ok)
{
    paktype *pak = NULL;

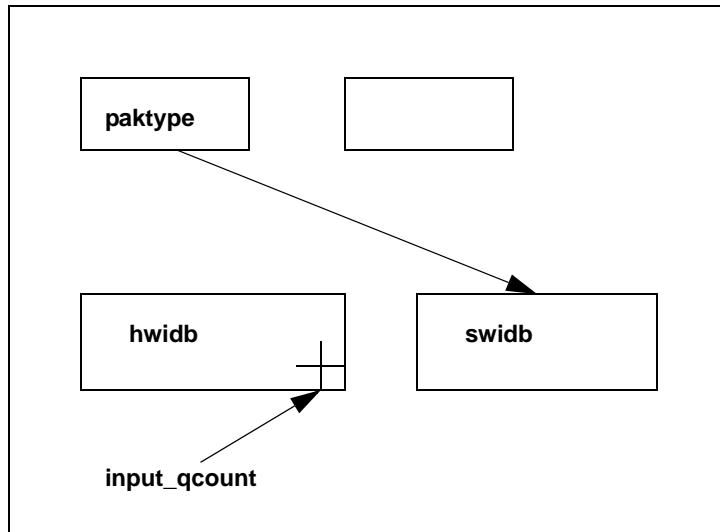
    /*
     * If cache_ok is set to be TRUE when the inline is used, this
     * code will be compiled in. *NO* conditional will be generated.
     */
    if (cache_ok) {
        /*
         * Try to get a cached buffer. if all cache buffers are
         * in use, get a buffer from the buffer pool.
         */
        pak = pool_dequeue_cache(myidb->pool);
    }

    /*
     * couldn't get a cache buffer, try the buffer pool.
     */
    if (!pak) {
        pak = pool_getbuffer(myidb->pool);
    }
    if (pak) {
        /*
         * got the packet, setup a pointer to the
         * start of the data. make sure it's longword
         * aligned (the chip requires this)
         */
        pak->datagramstart = pak->data_area + data_offset;
        (ulong)pak->datagramstart &= BUFFER_LONGWORD_ALIGN;
    }
    else
    {
        myidb->counters.output_nobuffers++;
    }

    return (pak);
}
```

5.3.16 Manipulate Input Interface for Packet Processing

It is important to be aware of the inbound interface while processing a packet. Figure 5-6 may help in understanding input interface manipulation.

Figure 5-6 Input Interface Manipulation

An interface is represented by an interface descriptor block, or IDB. A hardware IDB (HWIDB), or `hwidb` structure, is allocated at initialization time for each hardware interface as it is discovered, and contains common information about the interface such as the interface name, media type, state information, MTU size, port number, and pointers to packet-handling interrupt-level routines, as applicable. Each interface also has a software IDB (SWIDB), or `swidb` structure, associated with it. The SWIDB points to the related interface's HWIDB, and contains software protocol-related information about the interface.

When a packet is processed, its packet buffer header is associated with an input interface `idb`. Table 5-8 summarizes the input interface manipulation functions for packet buffers. These functions will set or clear the pointer in the packet header to the associated `idb`.

Table 5-8 Input Interface Manipulation Functions for Packet Buffers

Function	Purpose	Refer to
<code>set_if_input()</code>	Set the input <code>idb</code> pointer to the given SWIDB.	Section 5.3.17, “Associate a Packet Buffer with an Interface”
<code>input_getbuffer()</code>	Call <code>getbuffer()</code> , then set the input <code>idb</code> pointer from the given HWIDB.	Section 5.3.17, “Associate a Packet Buffer with an Interface”
<code>clear_if_input()</code>	Set the input <code>idb</code> pointer to <code>NULL</code> in the given packet buffer header.	Section 5.3.19, “Remove a Packet Buffer from an Interface”
<code>change_if_input()</code>	Replace the input <code>idb</code> pointer with the given new <code>idb</code> pointer.	Section 5.3.18, “Move a Packet Buffer to Another Interface”

The count of the number of packets in the network device attributed to this inbound hardware interface is stored in the HWIDB, in `hwidb->input_qcount`. The functions in Table 5-8 increment and decrement the `input_qcount` when a packet buffer’s association with an input queue changes.

5.3.17 Associate a Packet Buffer with an Interface

The Packet Buffer Services maintain a count of the number of packets in the system that are charged to a particular interface (also referred to as the interface's *input queue*, although it is not an actual queue structure). The input queue count is stored in the interface's HWIDB structure.

An interface's input queue count is incremented as follows:

- A packet buffer comes into the system via that interface.
- Ownership of a packet is transferred or moved to that interface.

The input queue count is important in balancing packet buffer resources between interfaces. If an interface's input queue count becomes too high, the interface is throttled.

To get a packet buffer from a public buffer pool and associate it with an interface's input queue, use the [input_getbuffer\(\)](#) function. This function takes a buffer data size and a pointer to the HWIDB for the input interface, obtains a packet buffer, and assigns the packet buffer header pointer to the SWIDB located via the given HWIDB.

```
paktype *input_getbuffer(int size, hwidbtype *idb);
```

This is equivalent to calling [getbuffer\(\)](#) followed by the [set_if_input\(\)](#) function, which is described next. This function is frequently used to obtain a contiguous buffer for coalescing a particle-based packet before sending it to process level.

To associate a new packet buffer with an interface when the packet buffer is obtained from a private buffer pool or cache, call the [set_if_input\(\)](#) function after obtaining the packet buffer yourself via [pool_getbuffer\(\)](#) or [pool_dequeue_cache\(\)](#). This function points the packet buffer header to the SWIDB for the input interface, which is passed in the `idb` parameter.

```
void set_if_input(paktype *pak, idbtype *idb);
```

When a packet is first received, the device driver will issue this function to indicate another packet is in the network device coming from this inbound interface.

Both [input_getbuffer\(\)](#) and [set_if_input\(\)](#) increment the interface input queue count in the associated HWIDB structure.

5.3.18 Move a Packet Buffer to Another Interface

Whenever a packet should be charged against an interface other than the one through which it arrived, it should be moved to that interface. Charging a packet buffer against an interface increments the interface's input queue count.

Moving a packet 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 packet buffer from one input interface to another, or to change the subinterface association of a packet buffer (in other words, associate the packet with a different SWIDB of the same HWIDB), use the [change_if_input\(\)](#) function. The `new_input` parameter is a pointer to the new IDB with which the packet buffer will be associated.

```
boolean change_if_input(paktype *pak, idbtype *new_input);
```

After verification, `change_if_input()` essentially calls `clear_if_input()` for the existing association, followed by `set_if_input()` for the new interface association. If the new interface's input queue is congested (queue count reached the threshold), the packet is not reassociated with the new input queue, so the packet is considered "dropped". The caller should always check the return value, and release the packet if necessary upon failure.

An example of transferring ownership of a packet buffer is in the entrance to a tunnel. A packet to be tunneled is transferred from the physical interface on which it arrived to a software-only or virtual interface that controls the tunnel.

5.3.19 Remove a Packet Buffer from an Interface

When a received packet has been partially processed and should no longer be charged against the interface, it should be removed from the interface's input queue count.

To remove a packet buffer reference from an input interface, call the `clear_if_input()` function, which safely decrements the interface's input queue count, and disassociates the packet buffer from that interface.

```
void clear_if_input(paktype *pak);
```

Call this function when a packet buffer is no longer needed, such as after a successful transmit, while waiting for further processing, or after a packet was dropped, but before the packet is returned to the originating packet buffer pool. The `datagram_done()` and `retbuffer()` functions internally perform a `clear_if_input()` when releasing a packet buffer.

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. The TCP processing code therefore calls `clear_if_input()` before saving the packet.

5.3.20 Manipulate Packet Buffers on Indirect Queues

During packet buffer processing, if a packet buffer might need to be on more than one queue, the Packet Buffer Services might use *buffer elements* for indirect queuing of packets. Indirect queuing, described in Chapter 22, "Queues and Lists", uses a *list element* to build the queue rather than the `pak->next` field. Packet buffer element resource use can be displayed with the **show buffers** command (`Buffer Elements` output fields), as described in Section 5.5, "Useful Buffer Commands".

Refer to Section 22.4, "Manipulate Indirect Queues", for details on using packet buffer indirect queue manipulation functions such as `pak_dequeue()`, `pak_enqueue()`, `pak_insqueue()`, `pak_requeue()`, and `pak_unqueue()`.

5.3.21 Packet Buffer Pools Code Example

The following code example shows a typical packet buffer pool implementation that creates a private packet buffer pool with a cache and assigns a fallback public packet buffer pool.

```
/*
 * Initialize private buffer pool if there isn't one.
 * Populate the pool, initialize a private buffer cache,
 * and populate the buffer cache with buffers from the newly
 * created private pool. Also set up the fallback public buffer
 * pool.
 */
if (!idb->pool) {
```

```
if (idb->pool_group == POOL_GROUP_INVALID)
    idb->pool_group = pool_create_group();

if (!idb->hw_namestring)
    idb_init_names(idb->firstsw, FALSE);

lance_buffer_pool = pak_pool_create(idb->hw_namestring,
                                    idb->pool_group, idb->maxdgram, POOL_SANITY | POOL_CACHE, NULL);
if (!lance_buffer_pool) {
    crashdump(0);
}
lance_buffer_pool->fallback = pak_pool_find_by_size(idb->maxdgram);
pool_adjust(lance_buffer_pool, 0, PRIV_NUM_BUF(ds->rx_size),
            PRIV_NUM_BUF(ds->rx_size), TRUE);

pak_pool_create_cache(lance_buffer_pool, PRIV_NUM_CACHE(ds->rx_size));
pool_adjust_cache(lance_buffer_pool, PRIV_NUM_CACHE(ds->rx_size));

idb->pool = lance_buffer_pool;
}
```

5.4 Particle-based Buffer Management

This section describes how to handle scatter-gather DMA packet data using particles.

5.4.1 Particles: Overview

The Cisco IOS software provides an extension to the packet buffer scheme called *particles* that allows network frames to be constructed from several data blocks rather than as one contiguous data block, as is expected in packet buffer management. Particles are discontiguous blocks of memory combined dynamically by the Cisco IOS network interface driver to hold a single packet.

Particle-based packets are often used in Cisco IOS driver architectures that need to receive data into and fast-switch with small, fixed-size blocks of memory.

Particles allow support for scatter-gather DMA schemes within drivers, which can result in 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 might be contained in three 500-byte pieces.

Currently, many platforms implement fast-switching in interrupt-level driver code using particle-based packet buffers. The alternative is using contiguous packet buffers, which is simpler but less memory-efficient. The particle-based implementation requires additional structures and a bit more CPU time to build and decompose packets. However, like the Packet Buffer Services, Particle Services are implemented on top of the generic pool code, which allows the use of all the generic pool features, such as dynamic growth and caching for more efficient buffer access.

The flow of implementing a particle-based buffer scheme is similar to that of contiguous packet buffers, including allocating and maintaining public and private pools of particles to hold packet data. However, currently most of the Cisco IOS process-level forwarding software expects that packet buffers are contiguous. As a result, particle-based buffers at interface driver level must be coalesced into a contiguous buffer before passing it to upper-level forwarding code (usually at process level) that is particle-unaware. This can impact process-level switching performance.

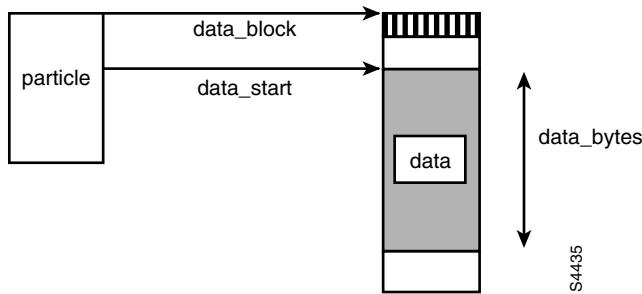
Upper-level forwarding and processing code *could* be redesigned to be particle-aware to help eliminate the need to coalesce particle-based buffers; however, the use of particles within driver and upper-level packet processing must be carefully designed.

5.4.2 Particle Structure

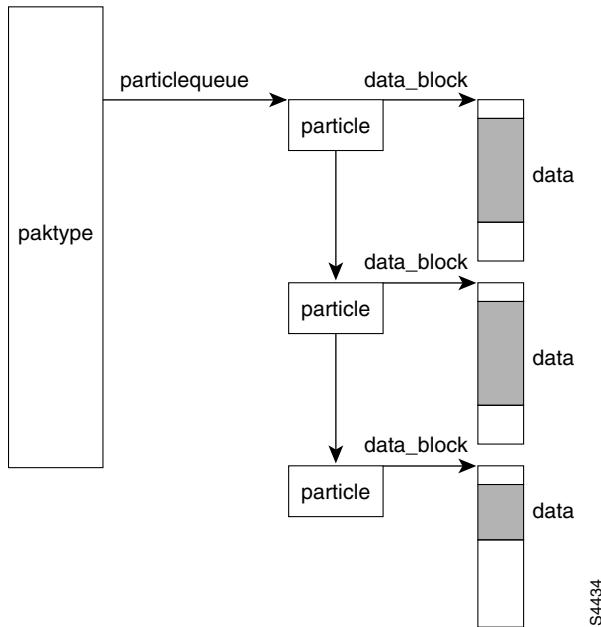
Particles consist of two fundamental blocks of memory, a particle header and an attached data block. The header is described by the `particletype` structure. Figure 5-7, which illustrates the structure of a particle, shows a particle with valid data of size `data_bytes` that starts at `data_start`. The `data_block` member 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.

As with packet buffers, some space is left before the usable data to allow room to rewrite a larger encapsulation. Similar to the packet buffer `ENCAPBYTES` and `TRAILBYTES` constants (see subsection 5.3.2, “Packet Buffer Data Blocks”), particles use `PARTICLE_ENCAPBYTES` and `PARTICLE_TRAILBYTES` constants to define the data area padding.

Figure 5-7 Particle Structure



A packet that has been divided into particles is maintained as a chain of particle headers, each pointing to a particle data block (also referred to as a *particleDB*). A packet buffer header is used to represent a particle-based packet, and maintains information about the entire packet. The `particlequeue` member of the `paktype` structure points to the first particle header in the chain of particles that form the complete packet, as illustrated in Figure 5-8.

Figure 5-8 Chain of Particles

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 using the `data_block`, `data_start`, `data_bytes`, and encapsulation and trailing padding, allowing flexible tailoring of the data considered to be actively part of the frame. When the frame data is being parsed, only the data indicated by the starting location and size in each particle header is considered valid. You need to make sure that these pointers and counts are updated, especially as encapsulations change.

Note that the `paktype` header has no data block of its own, unlike in contiguous packet buffers. Creating a packet buffer pool with a buffer size of 0 creates a pool of `paktype` headers suitable for use with particle attachments. Each particle header in the chain points back to the `paktype` packet buffer header that owns the chain.

5.4.3 Particle-based Buffer Pools: Overview

Particles are stored in pools in much the same way as contiguous packet buffers. The particle manager is implemented on top of the generic pool support and Packet Buffer Services, and allows the use of all the generic pool features, such as dynamic growth and pruning, and using pool caches. Particle pool clients use the Particle and Packet Buffer Services, and the generic pool management functions, to do the following:

- Create and maintain particle pools.
- Obtain particles and create chains to represent particle-based packets.
- Manipulate particle-based packets, coalescing them into contiguous packet buffers when required.
- Remove particles from particle chains, and return particles to their owning pools when done processing.

First the particle pool framework is initialized by calling `particle_pool_init()`, which sets up the list structures for maintaining public and private particle pools. This is usually called by `pak_pool_init()` at system initialization time when setting up packet buffer support, which also initially allocates the 6 public packet buffer pools, but does *not* usually allocate any particle pools. The actual allocation of public and/or private particle pools is generally done in the platform-specific initialization function, `platform_interface_init()`.

Table 5-9 summarizes the steps and API functions for creating and maintaining particle pools after `particle_pool_init()` has initialized the particle pool framework.

Table 5-9 Summary of Steps and Functions to Create Particle Pools

Step	Function	Description
Obtain a pool group number (if using a private particle pool).	<code>pool_create_group()</code>	Obtains a unique pool group number to create a private particle pool; pass group number to <code>particle_pool_create()</code> . Refer to subsection 5.2.7.1, “Pool Group Number and Creating Private Pools”.
Create the particle pool container.	<code>particle_pool_create()</code>	Allocates and initializes the <code>pooltype</code> structure for a new particle pool. Refer to subsection 5.4.4, “Create and Fill a Particle Pool”.
Create and add particles to the pool.	<code>pool_adjust()</code>	Repetitively calls the particle <code>create</code> function vector to create <code>permcount</code> permanent particles in the pool, and (for dynamic pools) additional temporary items to satisfy <code>mincount</code> . Refer to subsection 5.2.8, “Populate or Adjust a Pool”.
Associate a fallback particle pool with a private particle pool (optional, if using a private particle pool)	Assign the particle pool’s fallback member: <code>particle_pool->fallback</code>	Assign a backup pool to use when a particle pool is depleted. Refer to subsection 5.3.6.1, “Find and Assign a Fallback Packet Buffer Pool”.
Create a particle cache for a private particle pool (optional).	<code>particle_pool_create_cache()</code>	Allocates an array to associate a particle cache with a particle pool. Refer to subsection 5.4.5, “Create and Fill a Particle Cache”.
Populate a particle pool cache, if present (optional).	<code>pool_adjust_cache()</code>	Fills a particle cache array with particles from the associated particle pool. Refer to subsection 5.2.14, “Fill a Pool Cache”.

Table 5-10 summarizes the basic API functions to use particle pools and maintain particle chains in particle-based packets.

Table 5-10 Basic Functions for Particle Pool and Particle Chain Operations

Particle Operation	Function	Description
Obtain a particle from a public particle pool.	<code>getparticle()</code>	Returns a particle of a certain size defined by the <i>size</i> parameter from a public particle pool. Refer to subsection 5.4.6, “Obtain a Particle from a Particle Pool”.
Obtain a particle from a private particle pool.	<code>pool_getparticle()</code>	Obtains a particle from the particle pool specified by the <i>pool</i> parameter. Refer to subsection 5.4.6, “Obtain a Particle from a Particle Pool”.
Obtain a particle from a private particle pool cache.	<code>pool_dequeue_cache()</code>	Obtains a particle from the particle cache associated with the particle pool specified by the <i>pool</i> parameter. Refer to subsection 5.4.6, “Obtain a Particle from a Particle Pool”.
Lock a particle.	<code>particle_lock()</code>	Increments the reference count in the <i>particletype</i> particle header structure. Refer to subsection 5.4.7, “Manipulate the Reference Count of a Particle”.
Check the lock count of a particle.	<code>particle_get_refcount()</code>	Returns the reference count in the <i>particletype</i> particle header structure of a particle, which indicates how many code entities are currently using this particle. Refer to subsection 5.4.7, “Manipulate the Reference Count of a Particle”.
Release use of a particle when done processing it. This is equivalent to unlocking a particle.	<code>retparticle()</code>	Decrements the reference count on the particle in the <i>particletype</i> particle header structure. If the reference count reaches zero, returns the particle to its originating particle pool. Refer to subsection 5.4.8, “Return a Particle to a Pool”.
Add a particle to a particle chain.	<code>particle_enqueue()</code>	Adds the particle to the <i>end</i> of the particle chain attached to the packet header specified by the <i>pak</i> parameter. Refer to subsection 5.4.10, “Add a Particle to a Particle-based Packet”.
Remove a particle from a particle chain.	<code>particle_dequeue()</code>	Removes a particle from the <i>beginning</i> of the particle chain attached to the packet header specified by the <i>pak</i> parameter, and returns it to the caller. Refer to subsection 5.4.11, “Remove a Particle from a Particle-based Packet”.

Table 5-10 Basic Functions for Particle Pool and Particle Chain Operations (continued)

Particle Operation	Function	Description
Dismantle a particle chain and release the particles.	<code>particle_retbuffer()</code>	Dequeues and releases all the particles in a particle-based packet that are attached to the packet header without releasing the packet header itself. Refer to subsection 5.4.8, “Return a Particle to a Pool”.
Dismantle a particle-based packet, releasing the particles and packet header.	<code>datagram_done()</code> or <code>retbuffer()</code>	Dequeues and releases all the particles in a particle-based packet, and releases the packet header. Refer to subsection 5.4.8, “Return a Particle to a Pool”.
Convert a particle-based packet into a contiguous packet buffer.	<code>pak_coalesce()</code> or <code>particle_copy_to_buffer()</code> or <code>pak_copy_scatter_to_contiguous()</code>	Copy or coalesce a particle-based packet into a contiguous packet buffer. These functions have different levels of sanity checking and copy operation speed. Refer to subsection 5.4.12, “Copy a Particle-based Packet into a Contiguous Packet Buffer”.

Table 5-11 summarizes some of the available functions for manipulating particle-based packet data. Refer to the *Cisco IOS API Reference* for details on these and other available API functions.

Table 5-11 Summary of Particle-based Packet Data Manipulation Functions

Function	Particle-based Packet Data Operation
<code>pak_first_particle_size()</code>	Returns the data size of the first particle in the chain of a particle-based packet.
<code>particle_adjust_encap()</code>	Adjusts the size of the encapsulation padding within a particle-based packet.
<code>particle_center()</code>	Obtains the preferred location or “center” where the network header of a frame should start in a particle-based packet.
<code>particle_check_pak()</code>	Performs validity checking on a contiguous or particle-based packet.
<code>particle_clip_inline()</code>	Adjusts the data size within a particle-based packet.
<code>particle_head_clip_inline()</code>	
<code>particle_tail_clip_inline()</code>	

The next subsections describe in more detail the basic particle pool and particle-based packet functions referenced in Table 5-9 and Table 5-10.

Other subsections describe additional particle features and their supporting functions, such as:

- Packet Reparenting for Particles
- Particle Clones

5.4.4 Create and Fill 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);
```

Particle pools are maintained in public and private pool groups.

As for packet buffer pools, to create a public particle pool, use 0 for the `group` parameter to `particle_pool_create()`.

To create a private particle pool, call `pool_create_group()` (see subsection 5.2.7.1, “Pool Group Number and Creating Private Pools”) to obtain a unique group number and pass that to `particle_pool_create()` for the `group` parameter.

After initializing the pool container, fill the pool with available particles by calling the `pool_adjust()` function. The functions for particle pool operations are set internally in `particle_pool_create()` in the `particle_pool_item_vectors` function vector block. For details on parameter values, see the similar instructions for creating and filling packet buffer pools in subsection 5.3.5, “Create a Public Packet Buffer Pool”, and subsection 5.3.6, “Create a Private Packet Buffer Pool”.

5.4.5 Create and Fill a Particle Cache

To create a particle cache for a particle pool, call the `particle_pool_create_cache()` function. Particle pool cache operations are defined internally in `particle_pool_create_cache()` in the `particle_pool_cache_vectors` function vector block.

```
boolean particle_pool_create_cache(pooltype *pool, int maxsize);
```

After initializing the cache array, fill the cache with available particles from the associated pool by calling the `pool_adjust_cache()` function. For details, see the similar instructions for creating and filling packet buffer caches in subsection 5.3.14, “Create and Populate a Packet Buffer Cache”.

5.4.6 Obtain a Particle from a Particle Pool

Similar to obtaining a packet buffer, obtain a particle from a particle pool using the following guidelines:

- First, if available, attempt to obtain the particle from a private particle pool cache using the `pool_dequeue_cache()` function on the particle pool specified by the `pool` parameter.

```
void *pool_dequeue_cache (pooltype *pool);
```

- Next, if there are no particles available from a particle cache, attempt to obtain the particle directly from a private particle pool using `pool_getparticle()` for the private particle pool referenced by the `pool` parameter.

```
particletype *pool_getparticle(pooltype *pool);
```

- Finally, if there are no particles available from a private particle pool or cache, call `getparticle()` to get a particle from a public particle pool with particles of at least the size specified by the `size` parameter.

```
particletype *getparticle(int size);
```

5.4.7 Manipulate the Reference Count of a Particle

To lock a particle, which indicates a code element is actively using or processing the particle, call the `particle_lock()` function.

```
void particle_lock(particletype *particle);
```

Call the `particle_get_refcount()` function to check the reference count of a particle.

```
int particle_get_refcount(particletype *particle);
```

Do not attempt to decrement the reference count of a particle independently. Instead, use the `retparticle()`, `datagram_done()`, or `retbuffer()` functions to safely remove a reference to a particle (see subsection 5.4.8, “Return a Particle to a Pool”).

5.4.8 Return a Particle to a Pool

The functions in this subsection are considered safe methods to return a particle to its pool and decrement its reference count.

- To return a particle to a pool, use the `retparticle()` function.

```
void retparticle(particletype *particle);
```

- To remove all of the particles in a particle chain and return the particles to the pools from which they were obtained, without releasing the packet buffer header, `pak`, that held the particle chain, use the `particle_retbuffer()` function.

```
void particle_retbuffer(paktype *pak);
```

- To remove all of the particles in a particle chain, return the particles to the pools from which they were obtained, and also release the packet buffer header, `pak`, that held the particle chain, use the `datagram_done()` or `retbuffer()` functions. These functions checks whether the packet header has a particle chain and call `particle_retbuffer()` to return any attached particles.

Refer to subsection 5.3.9, “Return a Packet Buffer to a Pool”, for more information about the situations in which to use the `datagram_done()` or `retbuffer()` functions.

5.4.9 Prune a Public Particle Pool

As with packet buffer pools, over time, the particle pool manager will strive to reclaim unused particles from dynamic pools that contain more free particles than their defined `minfree` count. Periodically, the `particle_pool_periodic()` function runs at process level to call the `pool_prune()` function for any public and private particle pools in the system. This function returns particles that haven’t been used during the previous `POOL_PRUNE_TIME` interval. See subsection 5.2.8.1, “Pool Parameters in Dynamic Pools”, for more information.

All currently allocated particles constitute memory (usually IOMEM) that is “in use” as seen in `show mem` output, even when those particles are on their pool’s free list. If the system is short of memory, more aggressive pruning of free particles might be appropriate.

5.4.10 Add a Particle to a Particle-based Packet

To add a new particle to the end of the particle chain in a particle-based packet pointed to by `pak`, call the `particle_enqueue()` function.

```
void particle_enqueue(paktype *pak, particletype *particle);
```

The *particle* parameter points to the new particle to add to the chain.

5.4.11 Remove a Particle from a Particle-based Packet

To remove a particle from the beginning of the particle chain in a particle-based packet pointed to by *pak*, call the `particle_dequeue()` function.

```
particletype *particle_dequeue(paktype *pak);
```

The particle removed from the head of the chain is returned to the caller.

5.4.12 Copy a Particle-based Packet into a Contiguous Packet Buffer

Although most interrupt-level routines today deal with particle-based packets, some interrupt routines and all process-level code expect a packet to be contained in a single contiguous packet buffer. Therefore, before passing a packet to process level (or an output interrupt routine which does not support particles), the particle components of a packet must be copied or *coalesced* into a contiguous packet buffer so that other system components can correctly work with it.

Many platforms have DMA hardware that can coalesce scatter-gather DMA packets, as described in the *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code* manual in the subsection titled “Process-Level Contiguous-Only Considerations”. When this feature is not available, particle-based packet client software must perform this operation when required.

To coalesce a particle-based packet (shown here as *orig_pak*) into a contiguous packet buffer (shown here as *new_pak*), follow these steps:

Step 1 Acquire a new contiguous packet buffer of the correct size using one of the functions shown in the following examples, as appropriate:

- `new_pak = getbuffer(orig_pak->datagramsize)`
- `new_pak = input_getbuffer(orig_pak->datagramsize, idb)`

Step 2 Copy the contents of the particle-based packet (all particles in the chain) into the newly-obtained contiguous packet buffer using one of the following functions:

- `pak_coalesce()`

This function does sanity checking, recenters the packet in the new contiguous packet buffer, and copies only the actual data bytes in each particle of *orig_pak* to the destination packet buffer pointed to by *new_pak*.

```
paktype *pak_coalesce(paktype *orig_pak, paktype *new_pak);
```

The return value is `NULL` if the original particle-based packet or new contiguous packet buffer passed to the function is invalid, or either packet header does not contain a valid source pool pointer.

Note If the copy operation fails for any other reason, an error message is displayed, but the target contiguous packet buffer pointer, *new_pak*, is still returned. As a result, a non-`NULL` return value does not guarantee that the packet was successfully coalesced.

- `pak_copy_scatter_to_contiguous()`

This function copies a particle-based packet to a contiguous buffer, given the packet headers and starting locations for the data for both source and destination, and the packet data size in bytes to be copied.

```
void pak_copy_scatter_to_contiguous(paktype *pak, paktype *newpak,
                                    uchar *start, uchar *newstart, ushort size)
```

This function copies the relevant packet header members as well as the data, and assigns the data pointers in the destination. It does not adjust the starting location in the destination buffer to recenter the data, but simply uses the location passed in the `newstart` parameter. The `pak_coalesce()` function calls this function after calculating the ideal `network_start` center for the destination contiguous packet buffer, so use `pak_coalesce()` instead of this function if you want the data recentered when it is coalesced.

- [particle_copy_to_buffer\(\)](#)

This function does a fast copy (using `bcopy()`) of the entire particle data buffer contents of the particles in the particle chain in `scattered_pak` to the contiguous buffer location `ptr`.

It does not perform any sanity checking, or copy the source packet header fields to the destination packet header, or recenter or adjust the data in the destination. Use `pak_coalesce()` or `pak_copy_scatter_to_contiguous()` if you want these operations done with the copy operation.

```
void particle_copy_to_buffer(paktype *scattered_pak, uchar *ptr)
```

Step 3 Return the old particle-based packet buffer to the originating packet buffer pool using one of the functions shown, as appropriate:

- [datagram_done\(orig_pak\)](#)
- [retbuffer\(orig_pak\)](#)
- [particle_retbuffer\(orig_pak\)](#)

5.4.13 Packet Reparenting for Particles

Packet reparenting allows a particle-based packet to be associated with a new packet header structure, so the original `paktype` structure to which it was anchored can be reused.

An important situation in which reparenting is required is in interrupt-level packet forwarding of particle-based packets. In this case, the packets entering the network device are built off of a static `paktype` structure that is reused for each packet. When a packet is forwarded at interrupt level, it might not be placed directly on the Tx ring, but rather be enqueued to the outbound hold queue. The outbound hold queue expects packets to be anchored to a packet header structure, but the packet's static `paktype` structure will be reused by the Rx interrupt handler upon return of control. To safely hand off the packet to the outbound hold queue, a new packet header structure must be used for the packet for its new association with the outbound hold queue.

Packet reparenting addresses this issue, and is implemented by the [particle_reparent\(\)](#) function.

```
paktype *particle_reparent(paktype *pak, pooltype *pool)
```

The outbound driver code can call the `particle_reparent()` function, which does the following:

- Acquire a new `paktype` packet header structure.
- Move the particle chain to the new packet header, releasing the original static `paktype` structure.

- Migrate relevant members from the static `paktype` header to the new packet header.

The `pak` parameter to `particle_reparent()` points to the particle-based packet to be reparented, and the `pool` parameter references the packet header pool from which to obtain the new “parent” packet header.

For more information on reparenting, see the `particle_reparent()` API function, or consult various sections in the *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code* manual through the index entry “particle reparenting”.

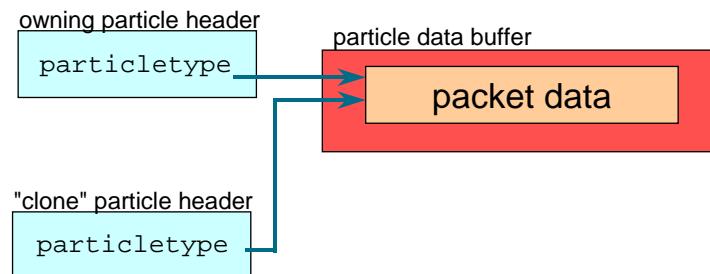
5.4.14 Particle Clones

Another feature of particles is *cloning*. Particle cloning allows a single particle data block to be logically identified as more than one distinct particle by assigning additional particle header structures of type `particletype` to point to it. The `particle_clone()` function implements particle cloning.

```
particletype *particle_clone(particletype *particle);
```

When a particle is cloned, the original particle header is considered to be the *owning particle header*, and the newly assigned particle header is the *clone particle header*. Both point to the same particle data block, so particle cloning does not involve any packet data duplication. The applicable members of the owning particle header are copied to the clone particle header, such as the data locations in the particle data buffer (`data_block`, `data_start`, `data_bytes`) and the associated packet buffer header (`pak`) to which the particle belongs. The clone particle header also maintains a pointer to the owning particle header (`original` member). See Figure 5-9 for an illustration of particle cloning.

Figure 5-9 Particle Cloning



Clone particle header structures are maintained in a system-wide clone queue. Any system components that clone a particle using `particle_clone()` or other general cloning API functions share particle headers from the same clone queue.

Table 5-12 summarizes other API functions for particle clones. Refer to the *Cisco IOS API Reference* for details on these and other available API functions.

Table 5-12 Summary of Other Particle Clone API Functions

Function	Description
<code>pak_clone()</code>	Clones a particle-based packet by assigning a new packet header, and cloning each particle in the packet's particle chain.
<code>pak_has_clones()</code>	Determines if a packet buffer has any cloned particles.
<code>particle_clones_add()</code>	Adds a specified number of particle headers and makes them available in the clone queue to use for creating particle clones. The <code>particle_clones_init()</code> function calls this to initially populate the clone queue with particle headers.
<code>particle_clones_init()</code>	Initializes and populates the clone queue with particle headers to use for creating particle clones. This function is usually called from the <code>platform_buffer_init()</code> function.
<code>particle_is_clone()</code>	Determines whether a particle header is the original owner or a clone.

A particle can be cloned multiple times, but a clone cannot be the owner of another clone. All clones that are created from the same particle have the same owning particle header. The `particle_clone()` function calls the `particle_is_clone()` function to check whether the particle header passed in the `particle` parameter is a clone, and if so, finds its owning particle header to which to assign the new clone's original member.

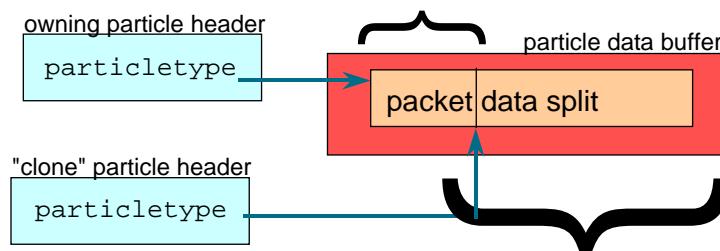
Usages of Particle Clones

There are two situations in which particle cloning is used.

One situation is to generate two particle queues to define a single physical packet, allowing for multiple usages, as in the following examples:

- For packet multicasting, to create two logically separate packets from a single chain of physical particle data buffers. After creation, each particle buffer can be manipulated independently.
- For UDP flooding, to duplicate the particle-based packet for each destination interface in the spanning tree.
- As support for Voice over IP "Lawful Intercept", duplicating packets to route to the Lawful Intercept server.

Another purpose for particle clones is to provide the logical separation of the data buffer into two separately managed areas. For example, Figure 5-10 shows a packet being fragmented and a clone being used to separate the data buffer into two logical buffers at the fragmentation location.

Figure 5-10 Using Particle Clones for Separately-managed Data Areas

5.4.14.1 Specifying a Clone Queue When Cloning Particle-based Packets

(New in Release 12.2s) Additional API functions have been added for cloning packets from a specified clone queue rather than a system-wide clone queue available to all system components. With the generally available system-wide clone queue, one system component can hog the particle headers in the clone queue and adversely affect the operation of other components competing for these resources.

Table 5-13 summarizes the API functions available for cloning particles using a particular clone queue rather than the system-wide clone queue. To support this feature, the `particletype` structure has an extra field indicating the clone queue from which the particle header was obtained if it was cloned.

Refer to the *Cisco IOS API Reference* for details on these and other available API functions.

Table 5-13 Particle Clone API Functions for Using a Specified Clone Queue

Function	Description
<code>pak_clone_with_cloneq()</code>	Clones a particle-based packet by assigning a new packet header, and cloning each particle in the packet's particle chain using particle headers from a specified clone queue rather than the shared system-wide clone queue.
<code>particle_clones_add_cloneq()</code>	Adds a specified number of particle headers to a specified clone queue for exclusive use of particle headers when creating particle clones.
<code>particle_clones_init_cloneq()</code>	Initializes and populates a specified clone queue for exclusive use of particle headers when creating particle clones.

To clone particles using a clone queue of your choice, follow these steps:

Step 1 Declare a structure of type `particle_clone_cb_type`, which represents your clone queue.

- Step 2** Call the `particle_clones_init_cloneq()` function to initialize your clone queue with details such as how many particle headers are available for cloning in this clone queue. Then this function populates the private clone queue with the specified number of available particle headers to use for particle cloning when you use the `pak_clone_with_cloneq()` function.
- Step 3** When cloning particle-based packets, use the `pak_clone_with_cloneq()` function and pass your clone queue for exclusive access to available clone particle headers.
- Step 4** To adjust the number of available particle headers for cloning in your clone queue, call the `particle_clones_add_cloneq()` function, specifying the clone queue to adjust and the number of cloning particle headers to add.

The following pseudo-code illustrates this new feature. To create and use a private clone queue that you do not want to share with other components in the system, initialize and declare your own clone queue variable, then call the `pak_clone_with_cloneq()` function to use your private clone queue.

```

/*
 * Global variable
 */
particle_clone_cb_type my_own_cloneq;
void my_private_pak_clone (paktype *original_pak)
{
    paktype *cloned_pak;
    /*
     *Clone the original_pak using my clone queue
     */
    cloned_pak = pak_clone_with_cloneq(original_pak, static_cloning_pak,
                                       &my_own_cloneq);
    broadcast_the_cloned_pak(cloned_pak);
}
/*
 * subsystem_init is done only once
 */
void my_application_subsystem_init (void)
{
    /*
     * Initialize my clone queue with 1000 particle headers all for me
     */
    particle_clones_init_cloneq(1000, &my_own_cloneq);
}

```

5.5 Useful Buffer Commands

For reconfiguring packet buffer and particle pool parameters from the command line, or displaying statistics about Packet buffer and particle pools on a system, you can use the **buffers**, **show buffers**, and **show buffer failures** commands.

In global configuration mode, the **buffers** command allows you to adjust the initial number of permanent and temporary buffers, as well as other dynamic pool free list parameters, for the preallocated public packet buffer pools. Use **buffers** in EXEC mode to tune other public and private packet buffer and particle pool parameters that are specific to particular interfaces (also called *interface* buffer pools in Cisco configuration documentation). Refer to Cisco product-specific configuration documentation for guidelines on adjusting buffer parameters at runtime using this command.

The **show buffers** command is often used for checking management of pools, packet buffers and particles, and is explained in the *Cisco IOS Configuration Fundamentals Command Reference, Release 12.4*. Similar documents can be referenced for other Cisco IOS release trains. In addition, the **show buffers failures** administration command is available to display a historical list of failures to acquire a packet buffer. (This feature does not exist for particles.)

Table 5-14 summarizes the **show buffers** user EXEC mode command output fields. This command displays information in the following order for all pools created in the system:

- 1 Public packet buffer pools
- 2 Private packet buffer pools
- 3 Public particle pools
- 4 Private particle pools

It also provides information on any associated pool caches.

See Table 5-16 and Table 5-17 for the **show buffers** enabled mode commands that are useful to developers for checking specific parts of packet buffers, particles, and pools in more detail.

Note The **show memory io enable** mode command displays all data buffers allocated from shared I/O memory.

Table 5-14 show buffers Command Output Fields

Output Field	Description
Buffer elements	Small structures used as placeholders for buffers in internal operating system queues. Buffer elements are used when a packet buffer might need to be on more than one queue. Detailed output follows of the fields described in the remainder of this table.
Name buffers	Heading naming the buffer pool for which detailed output follows of the fields described in the remainder of this table.
bytes	The defined size for each item in the pool.
total	Total number of packet buffers created by this buffer pool, which is the sum of <code>created</code> count and <code>permanent</code> count.
permanent	Count of permanent type buffers created.
peak	Highest number of packet buffers historically.
in free list	Total number of currently unallocated buffers.
min	Minimum number of buffers that should be present in the pool free list at any instant. For interface pools this will always be zero.
max allowed	Maximum number of buffers that can be present in the free list after usage for a while before being eligible for pruning.
hits	Count of successful attempts to allocate a buffer from a buffer pool free list.
misses	Count of unsuccessful attempts to allocate a buffer from a buffer pool free list, which would have resulted in growing the buffer pool.

Table 5-14 show buffers Command Output Fields (continued)

Output Field	Description (continued)
fallbacks	Count of buffer allocation attempts that resulted in fallbacks (to the smallest public buffer pool of size greater than or equal to the interface buffer pool).
trims	Count of buffers released to the system because they were not being used. This field is displayed only for dynamic buffer pools, not interface buffer pools, which are static.
created	Count of temporary type buffers created during the dynamic growth of the pool.
failures	Total number of allocation requests that failed because no buffer was available for allocation; the datagram was lost. Such failures normally occur at interrupt level.
no memory	Number of failures that occurred because no memory was available to create a new buffer.
max cache size	Maximum number of packet buffers from interface pool that can be in the buffer pool's cache. Each interface buffer pool has its own cache. These are not additional permanent buffers; they come from the interface's buffer pools. Some interfaces place all buffers from the interface pool into the cache. In this case, it is normal for the free list to display 0.
in cache	Number of items currently in the pool cache.
hits in cache	Number of times an item was successfully obtained from the pool cache.
misses in caches	Number of times no item was obtained from the pool cache.
buffer threshold	The low watermark for the pool cache. Whenever the number of items in the pool cache reaches this value, unthrottling is performed on the interface via the <i>threshold_callback</i> function registered by the driver.
threshold transitions	Number of times the low watermark for the pool cache was reached.

5.5.1 show buffers Output Example

The **show buffers** user-level command displays general information about indirect list elements used for building the list and also all pools created in the system.

The first section, **Buffer elements**, displays information about the buffer “elements”, which are used for indirect queuing of packets. Indirect queuing, described in Chapter 22, “Queues and Lists”, uses an intermediate “list element” to build the queue rather than using the first member of the structure, `pak->next`. Buffer elements are used when a packet buffer may need to be on more than one queue. This portion of the output contains the following information:

```
Buffer elements:
 499 in free list (500 max allowed)
 185520 hits, 0 misses, 0 created
```

- The maximum number of buffer elements that can be present in the free list, after their usage (`max allowed`) and the number of buffer elements currently available (`in free list`).

- The number of successful acquisitions from the list (`hits`), the number of unsuccessful acquisitions from the list (`misses`), and the number of temporary buffer elements created (`created`).

This is followed by a series of sections displaying information about each of the pools established and populated during system and interface initialization. Note that pools established with the `POOL_INVISIBLE` flag are not displayed.

```
Public buffer pools:
Small buffers, 104 bytes (total 50, permanent 50, peak 71 @ 1d16h):
    48 in free list (20 min, 150 max allowed)
    424256 hits, 125 misses, 51 trims, 51 created
    31 failures (0 no memory)

...
Private particle pools:
Serial1/0 buffers, 512 bytes (total 120, permanent 120):
    0 in free list (0 min, 120 max allowed)
    120 hits, 0 fallbacks
    120 max cache size, 42 in cache
    80 hits in cache, 0 misses in cache
    10 buffer threshold, 0 threshold transitions
```

The pools are displayed by type, as described in Table 5-15.

Table 5-15 show buffers Pool Types

Pool Type	Description
Public buffer pools	The six public packet buffer pools are created and populated during system initialization by the <code>pak_pool_init()</code> function and the <code>platform_buffer_init()</code> platform-specific routine, which is described in Chapter 7, “Platform-Specific Support”.
Interface buffer pools	Private packet buffer pools are created and populated during interface initialization, typically on a per-interface basis.
Header pools	Header pools supply standalone <code>paktype</code> structures for particle-based packets.
Particle clones	Particle clones are used to split a particle into two logical particles.
Public particle pools	As with public packet buffer pools, public particle pools are created for use by all interfaces and are established during system initialization.
Private particle pools	As with private packet buffer pools, private particle pools are created for use by a single interface and are established during interface initialization.

For each visible pool, the data provided is as follows:

1 General pool characteristics:

```
Small buffers, 104 bytes (total 50, permanent 50, peak 71 @ 1d16h):
```

The name of the pool (the first data in the first line, `Small buffers` in the example above), the defined size for each item (`bytes`), the number of packet buffers created by this pool (`total`), the number permanently allocated via the `pool_adjust()` function (`permanent`), and the highest number of packet buffers historically (`peak`).

The size may be adjusted by the pool-specific routine, for example:

- (a) For packet buffer pools (Public buffer pools and Interface buffer pools), each item consists of two elements, the `paktype` and the data buffer. The size of the data buffer is the input size incremented by `BUFFEROVERHEADBYTES`, or `sizeof(pakdata) + ENCAPBYTES + TRAILBYTES`.
- (b) For the Header pools, each item consists of just a standalone `paktype` structure.
- (c) For the Particle clones, each item consists of just a standalone `particletype` structure.
- (d) For particle-based pools (Public particle pools and Private particle pools), each item consists of two elements, a `particletype` header structure and the data buffer. The size of the particle data buffer is the input size incremented by `sizeof(pakdata) + PARTICLE_ENCAPBYTES + PARTICLE_TRAILBYTES`.

2 Information about the linked list:

`48 in free list (20 min, 150 max allowed)`

The number of items currently residing on the original linked list of items (`in free list`), and the minimum number of buffers that should be present in the pool free list at any instant (for interface pools this will always be zero), and the maximum number of buffers that can be present in the free list for a while, after their usage (`min, max allowed`). Items will be created or destroyed in an attempt to stay within this range, under the following conditions:

- (a) When running at process level, trims and creations are done during item retrieval and return. These routines are defined within the vector table for the pool, for example by `pak_pool_item_get()` for packet buffer and header pools and by `particle_pool_item_get()` for particle buffer pools.
- (b) When running at interrupt level, the pool is enqueued to the `pool_process` process for subsequent trims and creations using the `servpendingpoolQ` queue.

3 Information about usage and expansion/shrinking:

`120 hits, 0 fallbacks`

Or:

`424256 hits, 125 misses, 51 trims, 51 created
31 failures (0 no memory)`

The number of successful acquisitions from the list (`hits`), the number of unsuccessful acquisitions from the list (`misses` or `fallbacks`), the number of temporary type packet buffers destroyed (`trims`), the number of temporary type packet buffers created (`created`), the number of times no item could be returned (`failures`), and the number of times an item could not be created because the `malloc()` or `chunk_malloc()` memory allocation failed (`no memory`).

- (a) The term `fallbacks` is used for private pools with a fallback pool defined; `misses` is used for public pools and private pools with no defined fallback pool.
- (b) The statistics `trims`, `created`, `failures` and `no memory` are only displayed on pools with dynamic sizing, when the `POOL_DYNAMIC` flag is set.

Other information about these values can be viewed in the Cisco technical note, Document ID: 14620, “Understanding Buffer Misses and Failures”, at the following URL:

http://www.cisco.com/en/US/products/hw/modules/ps2643/products_tech_note09186a0080093fc5.shtml

- 4 Information about pool cache items, when a pool cache has been defined, via `pak_pool_create_cache()` or `particle_pool_create_cache()` and the `pool_adjust_cache()` functions:

```
120 max cache size, 42 in cache
 80 hits in cache, 0 misses in cache
```

The number of items defined for the pool cache (`max cache size`), the number of items currently in the pool cache (`in cache`), the number of times an item was successfully obtained from the pool cache (`hits in cache`), and the number of times no item was obtained from the pool cache (`misses in cache`).

- 5 Information about interface throttling characteristics:

```
10 buffer threshold, 0 threshold transitions
```

This specifies the low watermark for the pool cache (`buffer threshold`) because whenever the number of items in the pool cache reaches this value, unthrottling is performed on the interface via the `threshold_callback` function registered by the driver, and this specifies the number of times the low watermark was reached (`threshold transitions`).

5.5.2 show buffers Enabled Command Options

The **show buffers** enabled mode commands display additional information about packet buffers and particles on the system, as shown in Table 5-16.

Table 5-16 show buffers Enabled Command Options

Command	Description
show buffers all	This command runs through memory displaying information about each packet buffer and runs through memory again, displaying information about each particle buffer.
show buffers assigned	This command runs through memory displaying information about each packet buffer and particle buffer currently in use. Buffers that are assigned are defined as buffers that have the <code>refcount</code> equal to one or greater in their packet buffer or particle header.
show buffers free	This command runs through memory displaying information about each available packet buffer and particle buffer. Buffers that are free are defined as buffers that have the <code>refcount</code> equal to zero in their packet buffer or particle header.
show buffers old	This command runs through memory displaying information about each packet buffer and particle buffer for which the packet has remained longer than one minute. Buffers that are old are defined as buffers that have the input time equal to longer than one minute ago in their packet buffer or particle header.

Table 5-16 show buffers Enabled Command Options (continued)

Command (continued)	Description
show buffers input-interface <i>interfacetype/number</i>	This command runs through memory displaying information about each packet buffer and particle buffer that is associated with a particular interface.
show buffers address <i>address</i>	This command runs through memory searching for a packet buffer or particle buffer starting at the supplied address and then displaying information about it.
show buffers pool <i>pool-name</i>	This command runs through memory displaying information for each packet buffer or particle buffer assigned to the specified pool.

The **show buffers** command is subject to the modifiers shown in Table 5-17.

Table 5-17 show buffers Enabled Command Modifiers

Command	Description
show buf header	Display header structure information only. Indicated via SHOW_BUFFERS_DUMP_HEADER.
show buf packet	Display header structure information plus the packet. Indicated via SHOW_BUFFERS_DUMP_PACKET.
show buf dump	Display header structure information and the entire data buffer. Indicated via SHOW_BUFFERS_DUMP_ALL. Otherwise, just a single line is displayed for each packet buffer or particle buffer.

Interfaces and Drivers

Added new section 6.17 “Default Box-Wide IEEE MAC Address”. (April 2011)

Switched section 6.2 “Manipulate IDBs” and 6.3 “Subblocks and Private Lists”. Divided section 6.3 “Manipulate IDB Subblocks” into section 6.4 “Subblock Implementation Before Release 12.2S” and section 6.5 “Subblock Implementation After Release 12.2S”. (November 2010)

Added information related to HWIDB recycling to section 6.2.5 “Delete an IDB”. (June 2010)

Complete overhaul of section 6.2 “Manipulate IDBs” to document the new way of creating IDBs. Changed `pkt_scheduling@cisco.com` to `interest-idb@cisco.com` in note in section 6.1

“Introduction”. Changed `sys/h/subblock.h` to `subblock.h` in reference to where the globally assigned enumerator identifier is defined in section 6.3 “Subblocks and Private Lists”. (October 2009)

Added information to clarify the ways a developer can prevent the user from accessing or viewing an IDB. This is reflected in section 6.12, “Protecting an IDB from the User.” (November 2009)

6.1 Introduction

Note Cisco IOS interfaces and drivers questions can be directed to the `interest-idb@cisco.com` and `interest-os@cisco.com` mail aliases.

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.

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.

IDB information manipulation has evolved through the history of Cisco IOS software releases. This chapter includes sections describing how to migrate interface code to accommodate the significant implementation changes. Part of this evolution included the introduction of *subblocks*. Subblocks allow private variables and values to be associated with a hardware or software IDB without including all of this information in the basic IDB structure, for more modular, scalable, and memory-efficient access to the information.

Note The subsections in section 6.3, “Subblocks and Private Lists” and Section 6.4 “Subblock Implementation Before Release 12.2S” describe the subblock implementation in releases prior to 12.2S. Important changes in the subblock implementation for Releases 12.2S and above are documented in Section 6.5, “Subblock Implementation After Release 12.2S”; refer to this section, and the Cisco IOS API Reference, if modifying code or writing new code for accessing and manipulating subblocks.

This chapter includes information on the following topics:

- Manipulate IDBs
- Subblocks and Private Lists
- Subblock Implementation Before Release 12.2S
- Subblock Implementation After Release 12.2S
- Manipulate a Private List of IDBs
- IDB Data Structure Shrinking
- IDB Helper Functions
- Encapsulate a Packet
- Enqueue, Dequeue, and Transmit a Packet
- Getting and Setting IDB Fields
- Protecting an IDB from the User
- The linktype enum
- nextsub Subinterface List (new in 12.2T)
- Interface Locking Mechanism (new in 12.1)
- The Dev-Object Model
- Default Box-Wide IEEE MAC Address

6.1.1 Terminology

BM

Boolean or Bit Manager—IOS infrastructure to support dynamically allocated flags (Booleans) for IOS data structures.

FT

Function Table—a structure containing function pointers and other constant data, defining standard methods to be accessed from base class code. Each subblock has a pointer to a FT, defining the subblock type and pointers to the methods of the subblock class.

IDB

IOS Interface Descriptor Block—the IDB is a central data structure within IOS describing an interface. There are two flavours, Hardware IDBs (describing a physical connection or end point) and Software IDBs (also known as subinterfaces, describing separate virtual connections on a single physical connection).

Identifier

A value or handle used to identify a particular type of subblock on an IDB.

Subblock

Object attached to IDB containing private data owned by a subsystem.

6.2 Manipulate IDBs

Interface descriptor blocks (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 (IF-Index). 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.

This section describes how to create an IDB, link an IDB, iterate over a list of IDBs, delete an IDB, and reuse an IDB. The following functions all pertain to the use of `idb-identity` to create a unique identity for an IDB that can be used to identify that IDB on systems that employ redundancy. All physical IDBs should be created using `idb-identity` because redundancy is becoming pervasive throughout the product lines. SWIDBs are still created using `idb_create_subif()`. However, when they are deleted and if they may be reused for another interface, then `swidb_delete_identity()` should be called, and `swidb_set_identity()` should be called upon reuse. In other words, there are two models for interface deletion. The first model reserves the deleted IDB for reuse for that same interface should it return (OIR is an example of that). The second model clears the identity of the IDB and will reuse it for a possibly different interface. In the case of the second model, `swidb_delete_identity()` should be called when the interface is deleted, and `swidb_set_identity()` should be called upon reuse.

6.2.1 Terminology

idb-identity

The `idb-identity` is both a structure within the HWIDB that contains information to uniquely identify a HWIDB and also a component providing APIs for the creation or manipulation of IDB identities. Its use is recommended for interfaces that may be present on highly available redundant platforms.

IF-Index

The IF-Index is a generic term for the monotonically increasing numerical identifier within an IDB. For a SWIDB it is `swidb->if_number` and for a HWIDB it is `hwidb->hw_if_index`. The IF-Index is assigned to an IDB at creation and should never change.

6.2.2 Create an IDB

Two functions are provided to create 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.

There are convenience APIs that should be used in preference to the primitives when working with IDBs that are using the `idb_identity`. For IDBs that can be used on redundant systems, `idb_identity` should be used for creating the IDBs. For virtual IDBs, the VIDB should be considered.

```
hwidb_create_with_identity()
vidb_malloc()
vidb_malloc_with_identity()
```

To create a hardware IDB and its associated software IDB using an Identity, call the `idb_create_with_identity()` function. (*New in 12.2S*)

```
#include COMP_INC(idb, interface.h)
hwidbtype *idb_create_with_identity(idb_identity_t *identity);
```

The following pseudocode illustrates the use of the `idb_create_with_identity()` function:

```
void foo_create_idb (uint slot, uint unit)
{
    idb_identity_t identity;
    hwidbtype *hwidb;

    idb_id_clear(&identity);
    idb_id_set_type(&identity, IDBTYPE_FOO);
    idb_id_set_slot(&identity, slot);
    idb_id_set_unit(&identity, unit);

    hwidb = idb_create_with_identity (&identity);

    hwidb->snmp_if_index = reg_invoke_ifmib_register_hwidb(hwidb);

    idb_final_hw_init();
    idb_enqueue();
}
```

Basically you will have to call the following:

- 1) `idb_id_clear(&identity);`
- 2) `idb_id_set_XYZ(&identity, XYZ);`

For example:

- 2) `idb_id_set_type(&identity, IDBTYPE_FOO);`
- 3) `idb_id_set_slot(&identity, slot);`
- 4) `idb_id_set_unit(&identity, unit);`

And finally:

- 5) `hwidb = idb_create_with_identity (&identity);`

In some circumstances it is not possible to know the identity of an IDB prior to its creation, and in some cases an identity is never going to be assigned. To accommodate these rare situations, the `idb_create_blank()` function creates a blank API with no `hw_if_index` and no identity.

The following example code illustrates the creation of an interface where the identity is not known at the time of IDB creation and is added later prior to enqueue of the IDB to the IDB system queue:

```
hwidbtype eobc_idb;

void eobc_subsys_init (subsysstype *subsys)
```

```

{
    eobc_idb = idb_create_with_default_identity();

    eobc_init_idb();
}

void eobc_init_idb (void)
{
    idb_identity_t identity;

    idb_id_clear(&identity);
    idb_id_set_type(&identity, IDBTYPE_FEIP);
    idb_id_set_slot(&identity, 0);
    idb_id_set_unit(&identity, 0);

    hwidb_set_identity(eobc_idb, &identity);

    idb_final_hw_init();
    idb_enqueue();
}

```

6.2.3 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.

6.2.4 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.

6.2.5 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);
```

For interfaces that are using the `idb-identity` APIs, `hwidb_delete_identity()` for hardware interfaces and `swidb_delete_identity()` for software interfaces should be used at the time that the interface is deleted.

```
#include <idb-identity/include/idb_identity.h>
boolean hwidb_delete_identity(hwidbtype *hwidb);
```

6.2.6 Reuse an IDB

Hardware and software IDBs cannot be freed. Therefore, over time, the system could run out of allocable space for IDBs. For this reason, IDBs should be reused once deleted where possible. The IDB can be reused by either:

- 1 Using it for the same interface when that interface comes back (for example, OIRed).
- 2 Clearing the location of the IDB and reusing it for a different location, for example, Ethernet0/0.1 -> Ethernet0/0.2.

A common form of reuse is with OIR. When a card is removed, all HWIDBs associated with that card are deleted and references to them kept within a platform OIR structure. All configuration subblocks are retained on those interfaces. When a card of the same type is inserted into that same slot, the existing IDBs are reused, and the configuration is maintained.

Virtual IDBs (like tunnels) must also be reused. An application commonly keeps a reuse queue where deleted IDBs are stored and then recycled. There are a few points to remember when managing a reuse queue:

- 1 Call `hwidb_delete_identity()` or `swidb_delete_identity()` when an interface is deleted and put on the reuse queue. This will ensure that an IDB with that same identity can be assigned to a different IDB without causing a conflict to be detected.
- 2 On redundant systems, ensure that the Standby allocates IDBs with the same index that was used on the Active. Rewriting the index on an existing IDB is not allowed. So the reuse queue on the Standby should be searched for an IDB with the same index that was used on the Active. If none is found, a new IDB should be allocated on the Standby.
- 3 When an IDB is reused, `hwidb_set_identity()` or `swidb_set_identity()` should be called so that the new identity of the IDB can be tracked, and on redundant systems synced to the Standby unit.

The following is an example of reuse:

```
idbtype *foo_createidb (uint unit)
{
    idb_identity identity;
    idbtype *swidb;

    /*
     * Set up the identity of this application IDB
     */
    idb_id_clear(&identity);
    idb_id_set_type(&identity, IDBTYPE_FOO);
    idb_id_set_unit(&identity, unit);
```

```
/*
 * Attempt to recycle an existing IDB before creating a new one
 */
swidb = foo_recycle_idb(&identity);

if (!swidb) {
    /*
     * Could not obtain a recycled IDB, allocate a new one.
     */
    swidb = vidb_malloc_with_identity(&identity);
}

return (swidb);
}

static idbtype *foo_recycle_idb (idb_identity *identity) {
    idbtype *swidb = NULL;
    uint hw_if_index = IFNUM_ILLEGAL_INDEX;

    if (reg_invoke_rf_is_standby()) {
        hw_if_index =
reg_invoke_hwidb_if_index_retrieve_identity(identity);
    }

    swidb = find_matched_vidb_with_context(FALSE, (void*)hw_if_index,
foo_recycle_matcher);

    recycle_vidb(swidb);

    return (swidb);
}

static boolean foo_recycle_matcher (idbtype *swidb, void *context) {
    uint hw_if_index = (uint)context;
    boolean retval = FALSE;
    hwidbtype *hwidb = swidb->hwptr;

    if (is_foo(hwidb)) {
        /*
         * Have an old foo IDB, so reuse it if we can
         */
        if (reg_invoke_rf_is_standby()) {
if (hwidb->hw_if_index = hw_if_index) {
            /*
             * On Standby and have matched the same IDB as on the
             * Active, so reuse this IDB.
             */
            retval = TRUE;
        }
    } else {
        /*
         * On the Active choose the first available deleted foo IDB.
         */
        retval = TRUE;
    }
}
return (retval);
}
```

```
}
```

6.2.7 ATM IDB Recycling

ATM IDB recycling functionality enables an IDB to be reused by another ATM subinterface (p2p interfaces can be changed to p2mp links and vice-versa) after an ATM subinterface is deleted and all the Layer-2 subinterface parameters are cleaned up. The IDB cannot be reused by another type (non-ATM) interface; the IDB is not put back into the free pool and **show idb** does not show the IDB or the memory used by the IDB as being released after a deleting a subinterface.

Currently, IDB recycling is only available for ATM interfaces. Additional effort is needed to apply this to other interfaces, but the required IDB infrastructure is in Cisco IOS.

This functionality is only available in the 12.0S train.

6.2.7.1 Removal of ATM IDB Data Structures

All of the ATM-specific subblocks and linked data structures should be removed correctly at interface deletion. When a new ATM interface is created, the list of recyclable interfaces should be used to find an IDB. If this list is empty, a new IDB should be created.

For an ATM subblock, the following data structure needs to be cleaned up on interface deletion:

```
typedef struct atm_sw_sub_t_ {
    SWSB_BASE_LIST; /* ATM SW subblock header */
    struct wavl_handle_ *idb_vpivci_tree; /* vc_info tree on idb */
    idbtype *swidb; /* Ptr back to swidb */
    hwaddrtype *nsap; /* ATM Signalling NSAP Address */
    hwaddrtype *atm_e164_addr; /* ATM Signalling E164 Address */
    struct atm_arp_config_ *atm_arp; /* RFC 1577 ATM ARP data */
    mapclass_type *swidb_def_class; /* Default class for subinterface */
    .[ snip] ...

    atm_64bit_cntrs_t *cached_64bit_cntrs; /* cached copy of per-subif 64bit
                                                /* counters - used to respond to
                                                * queries less than 2s apart */
    atm_64bit_cntrs_t *cntrs_64bit; /* 64 bit octet and packet
                                         counters- aggregate of counters
                                         of deleted VCs */
    atm_64bit_cntrs_t *pre_clear_counter_cntrs_64bit; /* 64 bit octet and
                                                packet count aggregate
                                                * of counters of deleted VCs
                                                before clear counter cmd */

    .[snip] ...
    ulong atmsig_ubr_peak_rate; /* The peak rate to be used for
                                UBR on this subint */
    boolean map_filter; /* perform map filtering */
    tinybool atmsoft_check_ilmi;
    boolean rsvp_registered; /* To check if RSVP has
                                registered*/
    tinybool tbridge_trigger_svcs; /* flag to check if svc trigger
                                request is pending */
} atm_sw_sub_t;
```

6.2.7.2 Implementation Details for Enabling Layer-2 ATM Subblock Reusability

Broadly speaking, in addition to the reusability infrastructure already present in IOS, the following functionality has to be added by applications to support IDB reusability. Any Layer-2-specific encapsulation configuration structures in the subblock have to be cleaned up so that residual affects are not present when the subinterface is reused. In the case of ATM, it is achieved by the `atm_destroy_vc_list_on_subif()`, `atm_delete_vpvc_tree_from_idb()`, `atm_free_64bit_sb_cntrs()`, and `atm_free_config_defs_on_subif()` routines.

In addition to Layer-2-specific fields, the `idb->if_number` should be reset to 0 as part of the cleanup. So, platforms should register the cleanup routine with the `reg_invoke_platform_subif_cleanup()` API function provided by the IDB infrastructure.

Support has to be enabled on a per platform/interface/card_type basis. This is achieved through the `reg_invoke_is_subif_reusable()` registry. Currently, only ATM interfaces on the GSR platform are enabled.

6.2.8 IDB Protocol Counter API

The IDB Protocol Counter API is used to provide dynamic allocation of the hardware protocol counters. With dynamic allocation, a protocol's counters will be allocated only when used, which allows for memory savings. The counters were moved outside of the IDB, and thus, the counters are no longer visible from outside of the IOS Blob in IOS images. Registries are provided to modify the counters from outside of the IOS Blob.

The following hardware IDB protocol counter functions and registry functions were added to 12.2S:

- `idb_proto_counter_allocate()`
- `idb_proto_counter_decrement()`
- `idb_proto_counter_get()`
- `idb_proto_counter_increment()`
- `idb_proto_counter_reset()`
- `idb_proto_counter_set()`
- `reg_invoke_idb_proto_counter_get()`
- `reg_invoke_idb_proto_counter_increment()`
- `reg_invoke_idb_proto_counter_set()`

6.2.8.1 IDB Protocol Counter Functions

To create counters for a protocol on an IDB, call the `idb_proto_counter_allocate()` function.

```
#include "interface_private.h"
boolean idb_proto_counter_allocate(hwidbtype *hwidb,
                                    enum ACCT_PROTO protocol);
```

Protocol counters are automatically created on access. Calling this function will pre-allocate counters for the given protocol on an interface. If a protocol is never used on an interface, pre-allocation wastes memory. It is better to rely on the automatic allocation than to use this function. Use of the `idb_proto_counter_allocate()` function is strongly discouraged.

To get the value of a protocol counter, call the `edb_proto_counter_get()` function.

```
#include "interface_private.h"
counter_t idb_proto_counter_get(hwidbtype *hwidb,
                                enum ACCT_PROTO protocol,
                                enum PROTO_CNTRS type);
```

To increment a protocol counter, call the `edb_proto_counter_increment()` function.

```
#include "interface_private.h"
static inline void idb_proto_counter_increment(hwidbtype *hwidb,
                                               enum ACCT_PROTO protocol,
                                               enum PROTO_CNTRS type,
                                               counter_t size);
```

To decrement a protocol counter, call the `edb_proto_counter_decrement()` function.

```
#include "interface_private.h"
void idb_proto_counter_decrement(hwidbtype *hwidb,
                                  enum ACCT_PROTO protocol,
                                  enum PROTO_CNTRS type,
                                  counter_t size);
```

To set a protocol counter to a particular value, call the `edb_proto_counter_set()` function.

```
#include "interface_private.h"
void idb_proto_counter_set(hwidbtype *hwidb,
                           enum ACCT_PROTO protocol,
                           enum PROTO_CNTRS type,
                           counter_t size);
```

To reset the protocol counters on a HWIDB to an initial usable state, call the `edb_proto_counter_reset()` function.

```
#include "interface_private.h"
void idb_proto_counter_reset(hwidbtype *hwidb);
```

This function is called during IDB reuse and IDB creation, and thus, it is unlikely that any code outside of `interface.c` should ever need to call this function.

6.2.8.2 IDB Protocol Counter Registry Functions

Since it is quite common to set both packets and bytes, two counter get operations are supported by the registry functions, thus saving two IPC's. If you only need to get/increment/set one counter's value, this can be called using `IDB_CNTR_NONE` for the `type2` parameter. You still must supply a valid `size2` pointer even if `type2` is `IDB_CNTR_NONE`.

To get the protocol counter values from outside of the IOS Blob, call `reg_invoke_idb_proto_counter_get()`.

```
#include "../if/interface_registry.h"
static inline void idb_proto_counter_get(hwidbtype *hwidb,
                                         enum ACCT_PROTO protocol,
                                         enum PROTO_CNTRS type1,
                                         counter_t *size1,
                                         enum PROTO_CNTRS type2,
                                         counter_t *size2);
```

To increment the protocol counters from outside of the IOS Blob, call
`reg_invoke_idb_proto_counter_increment()`.

```
#include "../if/interface_registry.h"
static inline void idb_proto_counter_increment(hwidbtype *hwidb,
                                                enum ACCT_PROTO protocol,
                                                enum PROTO_CNTRS type1,
                                                counter_t size1,
                                                enum PROTO_CNTRS type2,
                                                counter_t size2);
```

To set the protocol counters from outside of the IOS Blob, call
`reg_invoke_idb_proto_counter_set()`.

```
#include "../if/interface_registry.h"
static inline void idb_proto_counter_set(hwidbtype *hwidb,
                                         enum ACCT_PROTO protocol,
                                         enum PROTO_CNTRS type1,
                                         counter_t size1,
                                         enum PROTO_CNTRS type2,
                                         counter_t size2);
```

6.3 Subblocks and Private Lists

Note The subsections in section 6.3, “Subblocks and Private Lists and Section 6.4 “Subblock Implementation Before Release 12.2S” describe the subblock implementation in releases prior to 12.2S. Important changes in the subblock implementation for Releases 12.2S and above are documented in Section 6.5, “Subblock Implementation After Release 12.2S”; refer to this section, and the Cisco IOS API Reference, if modifying code or writing new code for accessing and manipulating subblocks.

It is no longer necessary or desirable to store private variables in the main IDB structure. Use subblocks instead. For an example, see the Banyan VINES code. A subblock is a private data area that is attached to an IDB and is accessed through a globally assigned enumerator identifier defined in `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 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.

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 “Subblock Implementation Before Release 12.2S” and “Subblock Implementation After Release 12.2S”

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.”

All that is required to retrieve a subblock is the IDB pointer and a subblock identifier.

New features added to the router should use a subblock to hold feature variables, and Modular Interface Naming and Numbering should not add variables to the interface descriptors. Booleans should not be added to IDBs, but a dynamically allocated BM flag used instead.

6.3.1 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. Traditionally, 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++;
```

6.3.2 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.

6.3.3 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 Cisco IOS 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 lies on top of the Cisco IOS infrastructure.

6.3.3.1 Example: Creating a Subblock

The example in this section shows how a preallocated subblock is typically created, using AppleTalk as an example.

The `h/subblock.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. The example below is 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);
}
```

6.3.3.2 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.

Subblocks and Private Lists

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`. You need 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);
}
```

To retrieve a subblock pointer, 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.

```
subblockptr = idb_get_swsb(idb, subblock-identifier);
```

6.3.3.3 Use Subblock Reference Counters

To see that no other code is still using a subblock, use a reference counter rather than an access lock, so that when it comes to actually freeing the memory, it is only done when all the data structures that reference the subblock have stopped pointing to it—this avoids having hanging references. This way, the counter is not incremented every time you access the subblock, but every time you have another data structure set up a pointer to it. Since most subblocks are not referenced from anywhere else except the IDB, there is generally no need for a reference counter, but some applications may want to reference the subblock via some other data structure. For example:

```
idb_add_hwsb(...)

my_sb->refcount++; /* Since subblock is attached to hwidb */

...
my_other_data->sb_ptr = my_sb;
my_sb->refcount++;

...
idb_delete_hwsb(...);

if (--my_sb->refcount == 0) {
    free(my_sb);
}
...
```

```
my_other_data->sb_ptr = NULL;
if (--my_sb->refcount == 0) {
    free(my_sb);
}
```

So, every time someone sets up a reference to that subblock, the reference counter increments. Whenever someone stops referencing that subblock, the reference counter decrements, and when it hits 0, the subblock can be safely deleted.

6.3.4 Common Subblock Header

A 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.

6.3.4.1 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.

6.3.4.2 Subblock VFT

There is a virtual function table (VFT) pointer in each subblock header. This 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. 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. The main advantage of this 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.

6.3.4.3 Cisco IOS Virtual Functions versus IDB Function Vectors

When describing functions and function vectors, “virtual” has a very specific and defined meaning, especially since the original VFT concept copies the C++ virtual function idea, where *virtual* function is part of the language as a keyword. In Cisco IOS, we have taken the concept of virtual

functions used in C++ and implemented them in C. That is what a VFT is. Note that C does not have any natural support for virtual functions, but IOS has a set of conventions that we follow, and when we follow these rules, we can implement a similar concept to C++ virtual functions.

The idea of a C++ virtual function is that you have a base class that is used to represent something, and then have a derived class that includes the base class. The base class has a set of functions, and the derived class can override these functions with its own set of functions. Common IOS code can access the derived class to get to the base class and invoke the base class's functions, but the functions that are actually called are provided by the derived class.

In C++ you can have what is called an abstract base class, which is a base class that is only used as part of a derived class; the abstract base class can never be instantiated as an object. In the abstract base class, the functions can be typed as pure virtual, which means that the derived class must provide its own functions for these base class functions. Furthermore, the objects passed to the functions are treated as derived class objects, not base objects. (This is why in our VFT functions we always cast the first argument (the base object) to the derived object; normally C++ would do that for us.)

A good example in Cisco IOS is the plugin. The concept of a plugin is that it represents a port adaptor or some other physical card that can be plugged in. The common IOS code dealing with plugins expects certain functions to be provided, and a table of functions (a VFT) is used to call these functions. However, the VFT is supplied by the different drivers used to implement the real PAs. There is no such physical thing as a base plugin by itself. The plugin is an abstract class, which is only useful when a real plugin uses it as a base. Real port adaptors use the plugin as a base for building their own kind of thing, called a derived class. These derived classes can exist as real objects. For example, an Ethernet PA is a real thing, and there can be one or more of them existing in a system, but there is only one driver for all of them. Even though there are multiple Ethernet PA objects, there is only one class of Ethernet PA plugin. This Ethernet PA class is created by taking the base plugin and providing the functions via the VFT.

In other words, the common IOS code can treat the Ethernet PA plugin as a base plugin, without knowing any of the details of the plugin as an Ethernet PA plugin, and the common code calls the functions via the plugin's VFT. The VFT, however, is supplied by the Ethernet PA driver. But the common code doesn't know that, it just calls the functions via the base plugin. The functions called are virtual functions that are defined by the base class, but supplied by the derived class. The reason that IDB function vectors are not virtual functions is because IDB function vectors are not defined by a base class and supplied by a derived class. They are simply function vectors that are part of a data structure.

6.3.5 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->dialerdbs) {
        ddb = idb->dialerdbs;
        /* 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 FOR_ALL_HWIDBS traversal macro, and yet the client code does not have to maintain its own list of interfaces.

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.

6.3.6 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 access functions, there is little change to the bulk of the code.

6.3.6.1 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)
```

6.3.6.2 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.

6.3.6.3 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.

6.3.7 Address Filter Function Vectors

The address filter function vectors allow higher layer code to instruct the network interface driver to add and remove addresses for the hardware address filter table. There is also a routine to reset (sometimes used to refresh) the entire table(s). The hardware address filter table is normally located in the MAC controller of the network interface, that is, it allows the interface to do hardware-level filtering as opposed to software-level.

Most Ethernet, Fast Ethernet, and Gigether drivers in 12.3, 12.2, 12.2S, and 12.1E have implemented these vectors.

6.3.7.1 Benefits of Using These Vectors

Prior to the implementation of address filter function vectors, protocols and other higher layer code used to simply call `idb->reset` to initiate the addition/removal of an address from the address filter table. This caused unwanted resets/interface flaps in customer environments.

By implementing these vectors, code above the network driver is able to add, remove, and reset the hardware address filter table without doing a complete initialization of the driver and chip.

6.3.7.2 Address Filter Function Vectors Explained

- 1 `hwidb->listen()`—This vector is the same as `hwidb->afilter_add()`. (This is a legacy vector.)
- 2 `hwidb->listen_range()`—This vector is used to add a multicast address to the hardware address filter. `hwidb->afilter_add()` handles both unicast and multicast addresses. (This is a legacy vector.)

- 3 hwidb->afilter_add()—This vector is responsible for adding unicast and multicast (MAC) addresses to the hardware address filter. Typically, this vector maintains a shadow copy (software) of the table so that it can scan through the shadow copy to avoid multiple additions of the same address.

The address filter table has a limited number of entries, so this vector must also cater for an overflow of the table. In the scenario of an overflow, the network interface should be set to promiscuous mode. In this mode, we rely on software filtering of packets.

- 4 hwidb->afilter_remove()—This vector is responsible for removing unicast and multicast (MAC) addresses from the hardware address filter.
- 5 hwidb->afilter_reset()—This vector is responsible for refreshing/resetting the hardware address filter. It should clear all entries in the hardware and the shadow copy of the address filter table and then initiate the refresh. The refresh requires addresses to be re-added to the hardware address filter and is achieved by calling reg_invoke_media_interesting_addresses() on all the software IDBs.

Note The first entry in the hardware address filter should be the network interface's MAC address.

6.3.7.3 Example Driver

An example of a driver that implements these vectors is found in
/vob/ios/sys/pas/if_pas_i82543.c.

Here is some sample code from *if_pas_i82543.c*:

```
/*
 * i82543_init_idb:
 * IDB init routine for I82543 interfaces
 * also initializes instance structure
 */
static boolean i82543_init_idb (hwidbtype *idb, i82543_regs_t *base_addr)
{
    .
    .
    .

    idb->listen = i82543_add_hw_address;
    idb->listen_range = i82543_add_multicast_range;
    idb->afilter_add = i82543_afilter_add_wrapper;
    idb->afilter_reset = i82543_hwaf_reset;
    idb->afilter_remove = i82543_remove_hw_address
    .
}

}
```

6.4 Subblock Implementation Before Release 12.2S

Note The subsections in section 6.3, “Subblocks and Private Lists and Section 6.4 “Subblock Implementation Before Release 12.2S” describe the subblock implementation in releases prior to 12.2S. Important changes in the subblock implementation for Releases 12.2S and above are

documented in Section 6.5, “Subblock Implementation After Release 12.2S”; refer to this section, and the Cisco IOS API Reference, if modifying code or writing new code for accessing and manipulating subblocks.

6.4.1 Subblocks Types

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.

6.4.2 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.

6.4.3 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 `swhs_t`) as the first element in the private subblock structure. When calling the add function, pass the address of this common header.

Warning Because there are no checks to prevent a single subblock from appearing in multiple IDBs' lists, make sure that you do not add the same subblock to different IDBs.

6.4.4 Return a Pointer to an IDB Subblock

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);
```

6.4.5 Traverse a List of Subblocks

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)
```

6.4.6 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)
```

6.4.7 Release an IDB Subblock

When accessing a subblock with `idb_get_hwsb()` or `idb_get_swsb()`, you do not need to release the subblock when you are finished.

6.4.8 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.

When recycling a HWIDB, the driver is responsible for determining whether the information contained within the attached subblocks should be deleted. By convention, an interface retains its configuration even though the interface is not displayed during the configuration process. If the interface is enabled, its configuration will also be present. There may be some valid reasons suggesting not to do this all the time but that is the normal convention. The standard method of notifying that a HWIDB is being deleted(recycled) is to attach the `reg_invoke_swif_erase()` function to the component that contains the subblock and clean it via this service point. This function is called from the `delete_interface()` function when the HWIDB is being deleted(recycled).

6.5 Subblock Implementation After Release 12.2S

Note The subsections in section 6.3, "Subblocks and Private Lists and Section 6.4 "Subblock Implementation Before Release 12.2S" describe the subblock implementation in releases prior to 12.2S. Important changes in the subblock implementation for Releases 12.2S and above are documented in Section 6.5, "Subblock Implementation After Release 12.2S"; refer to this section, and the Cisco IOS API Reference, if modifying code or writing new code for accessing and manipulating subblocks.

(Available in Release 12.2S and above) In earlier releases, adding a new subblock to an IDB in Cisco IOS images required the following definitions:

- A subblock function table was defined in the subsystem's private area.
- A new subblock identifier was allocated in the global enumeration table in the file `h/subblock.h`.

The goal for the IDB Subblock Modularity changes was to allow any subsystem to allocate and install new subblocks on the IDB without having to modify generic or common code in any way, allowing new features and protocols to be added to IOS without modifying the existing IOS files.

The problem was that each subblock was referenced within the IDB by using an identifier that was statically allocated as a global enum in a common file. Managing this global list across all users who wished to use subblocks was problematic. If the addition of subblocks was done without this global allocation, then subblocks could be more widely used, since users would not be concerned about the problems of managing the identifier namespace.

Also, since the identifier namespace was global across all platforms and versions of IOS, many identifiers were not relevant for every IOS image, even though space in the IDB had to be allocated for the unused identifiers.

The goal was also to provide space in the IDB only for the subblock handles that are necessary, that is, actually used in the particular image in which they are built.

A secondary goal was to normalize the access and locking of the subblock structures. There was some inconsistency in the use of the macros to access subblocks, and this caused side effects, such as problems with removing subblocks, extra overhead in interrupt handlers, and memory holes. The

main confusion arose over the use of the `edb_use_hwsb()` and `edb_release_hwsb()` calls (both deleted in 12.2T), which incremented or decremented the usage count of the subblock. Many functions used these as some kind of access locking, when instead the usage count should have been a data structure reference count.

Other data structures, such as the CDB controller object, might find it useful to also have a similar subblock structure. Another goal of providing more modular subblock code is so it might be generic enough to be easily leveraged and reused for other IOS system components.

6.5.1 Subblock Identifier Changes

This section gives an overview of the differences in subblock identifier handling introduced with the new modularity changes. More details of these changes are included in subsequent subsections.

6.5.1.1 How Subblock Identifiers are Used

There are two uses of the subblock identifier in IOS, as follows:

- To provide an access handle so that given an IDB and a subblock identifier, the actual subblock can be accessed (using API functions such as `edb_get_hwsb()` and `edb_get_swsb()`).
- To obtain the head of the subblock list of all the subblocks of this particular type. Normally this is embedded as part of the `FOR_ALL_SWSB/FOR_ALL_HWSB` macros.

The standard API functions such as `edb_get_hwsb()` and `edb_get_swsb()` use the identifier as a handle to access the subblock via the IDB. All subblocks, when they are added to an IDB, are placed in either of the following lists in the IDB:

- A direct access “fast” array
- A linked list

Some classes of subblock need to be referenced from interrupt level or have time sensitive requirements, so the direct access array is provided in the IDB for a fixed number of subblocks. The identifier value, previously a global `enum` value that is now dynamically allocated, determines whether the subblock is referenced by a direct lookup in the subblock array, or by searching a linked list of subblocks on the IDB. A threshold value for the identifier defines whether the subblock is referenced via the fast subblock array in the IDB or a subblock linked list header in the IDB.

6.5.1.2 Modularity Solutions to Problems With Subblock Identifier Allocation

Subblock identifier allocation code in releases prior to 12.2S used the `HWIDB_SB_DYNAMIC` or `SWIDB_SB_DYNAMIC` constants as the thresholds to define whether a HWIDB or SWIDB subblock was referenced from the fast array or a subblock linked list. This overloading of the subblock identifier was not well known, so typically engineers would simply add a new `enum` value at the end of the existing `enum` list to define a new subblock type. Adding new subblocks that require fast access would increase the size of the IDB. The size increase affected all IDBs on all platforms, so the addition of new subblock types could have a significant unseen cost in terms of memory, even if the new types are only used on one platform.

Another use of the subblock identifier, to obtain the list header for the particular class of subblocks associated with the identifier, was done by using the subblock identifier as an index in an array of pointers (the subblock list headers). Every time a new subblock type was allocated, this array was expanded to include the new subblock list header. As an alternative, the subblock list header was moved to the module defining the subblock function table, and pointer to the list header stored in the function table.

Using a common global array to store the list headers was problematic. In a modularity sense, each list header belonged in the subsystem defining the subblock as private data, so one goal of subblock modularity was to move the list headers out of the common code into the relevant subsystem's data area. The scope of the list headers was extended to each piece of code that desired to iterate through the particular subblock list. However, with this change, the list headers are not available in global scope, so this needs to be taken into consideration when modularizing interface code.

The subblock lists were also singly linked. To provide for more scalable doubly linked lists in the future, the list head details were hidden inside a wrapper structure. Because a considerable number of subblocks do not even need to be linked together in a single list, the more modular structure saves the link pointer memory and the (potentially expensive) overhead cost of performing the linking.

6.5.2 Subblock Modularity/Scalability Changes

To summarize, the goals of the subblock modularity/scalability changes were as follows:

- Remove the global subblock enum identifier list.
- Move the list headers into each subsystem's private scope.
- Make the subblock lists optional on a per-subblock basis.
- Maintain the option of being able to quickly access subblocks through an array in the IDB.
- Allow the use of a large number of subblock types without impacting other platforms, features or protocols, effectively isolating the subblock allocation and making it independent of the general code.
- Allow new subblocks to be added without modifying existing structures or common code.
- Remove the access and reference count locking of the subblocks (via some API changes).
- Make the core data structures and code common for both HWIDB and SWIDB subblocks so that other data objects can reuse the infrastructure.

To accomplish these goals, the following changes have been implemented:

- Add three new entries to the subblock's function table:
 - A pointer to the list header structure for this subblock type, which is NULL if no list is required for this subblock type.
 - A pointer to an integer value where a dynamically allocated identifier is stored.
 - A pointer to the function to use to add the subblock into its owning subblock list. This facilitates adding a subblock to its list in IDB-sorted or non-sorted order.

The memory referenced by these pointers should be permanently allocated; that is, do not pass the address of a stack-local variable.

- Remove the enqueue and unlink function table entries and instead rely on the existence of the list header pointer to indicate whether the subblock is to be placed on a list.
- Move the subblock list link pointer out of the base structure and make it optional.
- For each subblock function table, add the new pointers and create new variables that are referenced from the function table.
- Remove the enum list, and implement an automatic mechanism for assigning a unique subblock identifier for new subblock types.
- Modify the FOR_ALL_HWSB/FOR_ALL_SWSB macros to reference the list header as a direct reference rather than using the old identifier as an index to a global array.

- Remove the usage counters.
- Normalize the use of the access routines and remove the usage counter access functions.
- Collapse the separate HWIDB and SWIDB data structures into a common core subblock structure and function table.

The changes required to most of the code are minimized by naming the new subblock identifier variables for existing subblock types using the same uppercase names previously used in their statically-assigned enum constant identifiers. For example, a subblock identifier for AppleTalk was defined in the previous subblock implementation as:

```
typedef enum {
    ...
    SWIDB_SB_APPLE,
    ...
} swidb_sb_t;
```

For the new implementation, other code changes are minimized by defining the subblock identifier variable as:

```
uint SWIDB_SB_APPLE;
```

If the new identifier variable names are the same as the original enum constant names, other code referencing the subblock using the standard API and inline functions does not change. However, subblock identifiers defined in *new* code should use lower case variable names according to the convention for variable names, rather than the upper case names as used for compatibility with previous enum values.

6.5.3 Subblock Modularity Data Structure Changes

The data structures used to represent subblocks have some changes to accommodate the modularity/scalability goals.

6.5.3.1 Subblock Function Table

Both SWIDB and HWIDB subblocks now share the same subblock function table structure, which is as follows (elements such as hwsb_enqueue and hwsb_unlink have been removed):

```
typedef struct sb_ft_ {
    uint *sb_type;
    sb_list_t *sb_list;
    sb_void_t sb_link;
    sb_bool_t sb_destroy;
    char *sb_name;
    /* The following field(s) are not necessarily statically
     * initialized in code which defines sb_ft structures. They
     * can therefore be NULL. Please leave all such fields at
     * the structure's end.
    */
    sb_void_t sb_unlink;
} sb_ft;
```

Table 6-1 describes the structure elements.

Table 6-1 Subblock Function Table (sb_ft) Structure Elements

Structure Element	Description
sb_type	Pointer to an unsigned integer into which the subblock support code stores the dynamically allocated subblock identifier. The allocation is done once for each subblock class, the first time that a subblock of this type is added to an IDB.
sb_list	Pointer to a list head structure for the subblock list (if no subblock list is to be maintained for this subblock class, this is NULL).
sb_link	Specifies the function used to link the subblock into the subblock list. If sb_list is NULL, this field must be (sb_void_t) return_nothing. If sb_list is non-NULL, then this field must be initialized to the [hs]wsub_link() function for subblocks linked in IDB sorted order, or the sb_append() function (shared for hardware and software IDBs) for subblocks not linked in IDB sorted order.
sb_destroy	Specifies the function to remove and delete a subblock from its list in an IDB. This element must be initialized to one of the following functions: [hs]wsub_destroy() for hardware or software IDBs, respectively, that can use the standard destroy function. (sb_bool_t) return_true if a component does not delete its subblocks. However, note that in general this is <i>NOT</i> advised. A private, non-standard destroy function, if more cleanup is necessary than provided by the standard destroy function. This function should include a call to the standard destroy function, [hs]wsub_destroy().
sb_name	Specifies a string name for the subblock.
sb_unlink or other fields	Other functions used by a subblock, if needed, should be defined at the end of the sb_ft structure.

A typical subblock function table would be assigned as follows:

```
uint HWIDB_SB_SERIAL;
sb_list_t serial_sb_list;
const sb_ft serial_ft =
{
    &HWIDB_SB_SERIAL,
    &serial_sb_list,
    hwsb_link,
    serial_sb_destroy,
    "SERIAL SB"
};
```

Naming the subblock identifier variable HWIDB_SB_SERIAL, which is the same as the constant name used in the earlier release's enum list, means that all the code referencing the subblock using code such as the following will not need to be modified:

```
serial_sb *ssb;
...
ssb = idb_get_hwsb(hwidb, HWIDB_SB_SERIAL);
```

6.5.3.2 Assign Subblock Identifiers Dynamically with an Indirect Identifier Field

The subblock support code that adds the subblock to the IDB (`idb_add_swsb()` or `idb_add_hwsb()`) tests the identifier value, and if it is zero (as it will be the first time that a subblock of this type is added to the system), a new, unique subblock identifier value is assigned to the identifier variable using a global static index variable.

To allow this dynamic subblock identifier assignment, the subblock identifier field in the subblock function table structure is a *pointer* to the identifier variable (`uint *sb_type`) rather than an identifier value itself. Code that attempts to access subblocks on an IDB of a type that has not yet been added will provide a zero value as the identifier, which is reserved by the system (a NULL value is returned, as expected). The next subsection describes the structure and constant changes that support this implementation.

6.5.3.3 Assign Subblocks in Fast Array or Subblock List

Depending on the initial value of the subblock identifier, the new common subblock code distinguishes between those subblock types that should use fast access via the IDB array vs. those that should use a subblock linked list, and dynamically assigns an identifier value accordingly.

To indicate whether to access a subblock in the fast array or linked list in an IDB, before adding it to the IDB, first initialize the subblock identifier variable with either of the `_ACCESS` constants shown below from `cisco.comp/idb/include/subblock.h`:

```
/*
 * Subblock IDs allocated up to this number are accessed thru
 * the fast array.
 */
#define HWSB_FAST      40
#define SWSB_FAST      32
...
/*
 * Default for index values.
 */
#define SB_LIST_ACCESS    10000 /* Don't install in fast array */
#define SB_DEFAULT_ACCESS 0      /* Default access for subblock */
```

The following subsystem code example initializes the subblock identifier to `SB_LIST_ACCESS`, indicating to insert this subblock in the IDB subblock linked list rather than using the fast access array:

```
uint HWIDB_SB_MY_SUBBLOCK = SB_LIST_ACCESS;
```

The subblock identifiers accessed from the fast array will be assigned values between 0 and `HWSB_FAST` or `SWSB_FAST` for HWIDB or SWIDB subblocks, respectively, so the identifier is the index into the array. Subblocks accessed through a search of the subblocks on the IDB will have values greater than `SB_LIST_ACCESS` based on a statically incremented index, and are located by comparing the type identifier values when traversing the list.

The default access is via the fast array, because usually there are fewer subblocks added to an IDB than `HWSB_FAST` or `SWSB_FAST`. However, some subblocks have explicit pointers in the IDB (e.g. FR, ATM), so the allocation of an extra pointer in the IDB is not required. Longer term, it is desirable to remove these explicit pointers, but considerable code change may be required to do this.

The new base subblock structure is as follows:

```
struct sb_t_ {
    struct sb_t_ sb_chain;
    union {
```

```

        struct hwidbtype_ *hwidb;
        struct idbtype_ *swidb;
    } sb_backp;
const struct sb_ft_ *sb_ft;
} ;

```

With the use of a common subblock structure for HWIDBs and SWIDBs, the back pointer to the owning object becomes a union. (For future objects, this could be a `void *` that is cast by an appropriate access function).

As the new base structure shows, the subblock list link pointer has been removed from the base structure. Either of the following base structure macros should be added explicitly *after* the base subblock definition to only include link fields if needed for linked list subblocks:

```

#define SWSB_BASE      sb_t swsb_base
#define HWSB_BASE      sb_t hwsb_base

#define SWSB_BASE_LIST sb_t swsb_base; sb_t *sblist_next; sb_t *sblist_prev;
#define HWSB_BASE_LIST sb_t hwsb_base; sb_t *sblist_next; sb_t *sblist_prev;

```

This provides a memory saving for subblocks that are not linked on a common list.

6.5.3.4 Subblock List Header Structure

To allow for future scalability changes on the subblock list handling, the subblock list header has been defined as the following structure of type `sb_list_t` to accommodate singly or doubly-linked lists without having to change the common code:

```

typedef struct sb_list_ {
    sb_t *head;
    sb_t *last_insert_hint;
} sb_list_t;

```

6.5.4 API Changes

The following changes were implemented in the interface API subblock functions and macros:

- The arguments to the subblock access routines now accept a `uint` for the subblock identifier instead of the subblock enum type.
- The functions `edb_swsb_first()`/`edb_hwsb_first()` were removed. The arrays holding the list headers were also removed.
- The `FOR_ALL_HWSB` and `FOR_ALL_SWSB` macros now both use a common macro, `FOR_ALL_SB`, for iterating a subblock list. The type identifier was replaced with a reference to the actual list header; any code iterating a particular subblock list must have the list header pointer within its scope, or have some way of obtaining its value.
- The functions `hwsb_unlink()` and `swsb_unlink()` were replaced by a single function `sb_unlink()`.
- The functions `hwsb_link()`, `hwsb_link_in_idb_order()`, `swsb_link()` and `swsb_link_in_idb_order()` were collapsed into two functions `hwsb_link()` and `sbsb_link()` that add the subblock in the correct IDB order. The default for `sbsb_link()` is `(sb_void_t) return_nothing`.
- A NULL check was added for `sb->sb_ft->sb_link`.

- Functions that referenced the types `hwsb_t` and `swsb_t` now only reference the single type `sb_t`.
- The functions `swsb_delete()` and `hwsb_delete()` do not return a boolean, since they will always succeed.
- Code should use the access functions, `sb_to_hwidb()` or `sb_to_swidb()`, to obtain the back pointers to the owning HWIDB or SWIDB rather than using the `sb_backp` union pointers directly.
- Standard destroy functions `hwsb_destroy()` and `swsb_destroy()` were added to perform the combined action of removing a subblock from its subblock list or the fast array, removing it from its owning IDB, and freeing the subblock.

6.5.5 Add a New Subblock with Modularity Changes

Adding a new subblock is straightforward, and with the change to migrate the subblock identifier from a common `enum` list to a variable within a subsystem, there are no changes required in common code to add a new subblock.

The following example adds support for a subblock designed to be attached to a HWIDB:

```

uint my_sub;
const sb_ft my_sub_ft =
{
    &my_sub,
    NULL,
    (sb_void_t)return_nothing,
    my_sub_destroy,
    "MY SB"
};

...
static boolean my_sub_destroy (sb_t *sb)
{
    /*
     * component-specific custom subblock cleanup operations go here
     */
    ...
    hwsb_destroy(sb);
    return (TRUE);
}

Header file:
...
typedef struct {
    HWSB_BASE;
    ...
} my_sub_t;
extern uint my_sub;

extern sb_list_t my_sub_list;
#define FOR_ALL_MY_SUB(sb)    FOR_ALL_HWSB(sb, my_sub_list)

```

With the appropriate naming substitutions of `sw/SW` for `hw/HW`, these examples could apply to subblocks for a SWIDB.

The example shows only a small number of places where specific references are made to the hardware or software IDB, so subblocks and their supporting code could easily be migrated between HWIDBs and SWIDBs.

6.5.6 Modularity Scope Issues

With the conversion of the subblock identifier from a global enum to a variable, there arises the question of where the variable should be declared and defined. Any code that referenced the subblock previously simply used the enum value declared in subblock.h, but now all code that references the subblock requires an extern declaration of the subblock identifier, as well as runtime link access to the variable itself. This is a tighter modularity requirement than previously, and forces some consideration of subsystem and link image modularity.

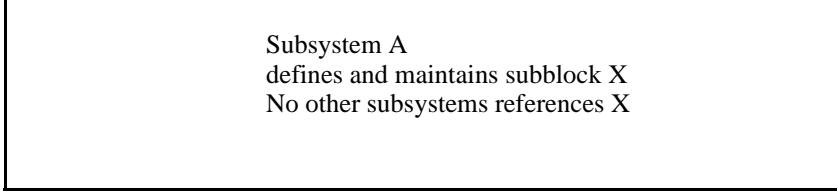
One way to deal with this is to declare the variable in the same header file where the subblock itself is defined, and define the variable where the subblock function table is defined. In some cases this does not always work, since other subsystems may attempt to reference the subblock without the supporting code being linked into the image. In this circumstances, it is useful to have a separate file defining the subblock index (and optionally the subblock list header) and include this in images that require it.

It is tempting to define the subblock indices in a globally included file such as interface.c or subblock.c but this *should be avoided* if at all possible, and instead the core modularity issue solved.

Another approach that is sometimes used is to define a set of STUB registries to access functions that are optionally in the image; again, this *should be avoided* since the STUB registries are simply trading modularity for a global registry, and do not solve the problem.

Most of these modularity situations fall into one of three categories:

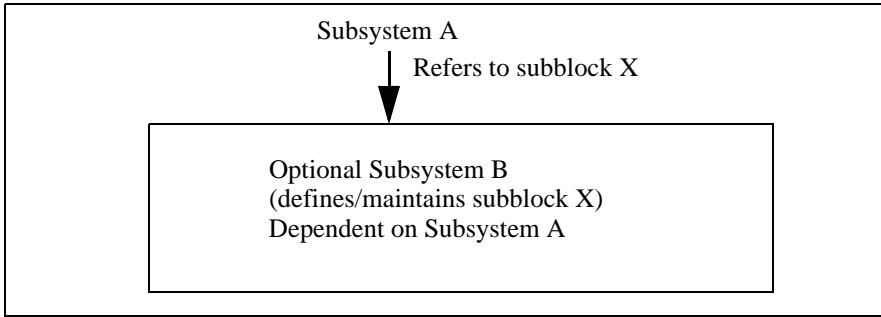
- Subsystem is self-contained:



Subsystem A
defines and maintains subblock X
No other subsystems references X

This is the simplest and easiest situation to deal with. Subsystem A simply defines the index variable to be part of its data area.

- Subsystem or core code accessing an optional and dependent subsystem's subblock:



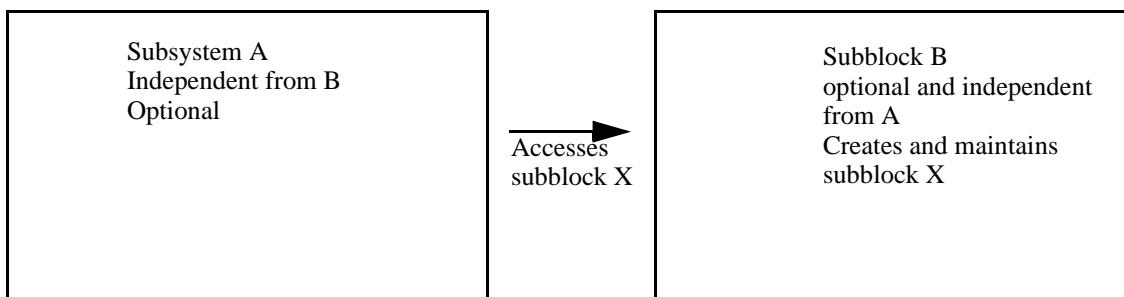
Subsystem A
↓ Refers to subblock X

Optional Subsystem B
(defines/maintains subblock X)
Dependent on Subsystem A

In this instance, subsystem B is dependent upon A i.e. if B is in a image, A must also be included. Subsystem B creates/maintains subblock X, but A refers to X (and if X does not exist, this is an indication to A that the feature is not enabled or not included). An example of this would be IP Multicast, which obviously relies on the IP subsystem, but Multicast may be an optional component of the image.

For this situation, a reasonable solution is for subsystem A to contain the subblock index variable for subblock X, so that it can refer to it if required, but it is also accessible to B if that subsystem is included. If both A and B are excluded from the image, then there is no requirement for the subblock index to exist anywhere in the image.

- Subsystem accessing an independent and possibly optional subsystem's subblock:



This situation is less common, but occurs when optional subsystems need to access data in subblock X, which itself may be optional; subsystem A may determine whether the data is available by checking for the existence of X, but to do that it needs access to the index variable for subblock X. It is not sufficient to place this variable in A, since A itself may not exist in the image, and it cannot be placed in B, since it may be optional as well.

There are a couple of solutions that are desirable in this situation; one is to determine both A and B depend upon a common subsystem, e.g. if A were IP multicast, and B were IP CEF switching, then both A and B depend upon the common core IP subsystem, so the variable can be placed there. This is good if both have a common dependent subsystem.

Another solution is to create a separate API object module that contains the index variable (as well as any other core API functions) and then explicitly include this in images where A is present but B is not. This works well, but it can be difficult to discover all the instances where this is required, and is no guarantee that as module and image changes are made, the appropriate changes are made.

A better and more reliable solution is to place the API object module into a library which is then linked against so that the module is automatically included if it is referenced. This is the preferred solution for this particular situation (though there are not many instances of this).

The modularity issues raised when subblock index variables are required to be accessed by independent subsystems is useful to catch the dependency issues, and can be very useful to show exactly what the subsystem dependencies are—in some cases, this may highlight design flaws, or to show where modularity problems are occurring.

6.5.7 Guidelines to Convert to New Subblock Implementation

To convert an existing subblock to use the new infrastructure, use the following guidelines:

- Declare a `uint` variable for the subblock index in an appropriate header file.

- Rework the subblock function table to reflect the new `sb_ft` type, remove the link/unlink vectors, and include pointers to the subblock index variable and optionally the subblock list header.
- Change `idb_use_[hs]wsb` references to use `idb_get_[hs]wsb` instead.
- Remove any calls to `idb_release_[hs]wsb()`.
- Because the usage counts have been removed, any code that is relying on the usage counts for locking should use a different mechanism, such as an explicit lock or usage count *in* the subblock, or `mem_lock()`/`free()` calls to automatically remove the memory when it is no longer being used.
- If the subblock destructor calls the `[hs]wsb_delete()` or `idb_delete_[hs]wsb()` functions, remove any checks on the return values because these functions no longer return a boolean.

If the subblock access is via a linked list, make the following changes:

- Use `[HS]WSB_BASE_LIST` instead of `[HS]WSB_BASE` in the definition of the subblock structure to add the subblock list pointer(s).
- Define a subblock list header and initialize the list header pointer in the subblock function table to point to it. Declare this list header in an appropriate header file; it must be accessible to any code attempting to iterate this subblock's list.
- Define and use a private subblock linked list traversal macro that calls the `FOR_ALL_[HS]WSB` macro with your list header, rather than calling the base `FOR_ALL_[HS]WSB` macro directly, to hide the name of the list header. For example:

```
#define FOR_ALL_SERIAL_SB(sb) FOR_ALL_HWSB(sb, serial_splist)
```
- Prior to the subblock modularity changes, a subblock identifier had a defined value before the subblock was created. The modularity changes removed the `enum` list that previously defined identifiers for all subblocks; a subblock identifier is not assigned until the subblock is first created and added to an IDB. Care should be taken to modify code that assumed a static subblock identifier value. This issue might commonly exist with ION subblock implementations.

6.6 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.

6.6.1 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 `edb_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);
```

6.6.2 Add an IDB to a Private List

To add an IDB pointer to a previously created private IDB list, use the `edb_add_hwidb_to_list()` or `edb_add_swidb_to_list()` function. The interface specified is inserted into the list in ascending unit-number order.

```
boolean idb_add_hwidb_to_list(struct hwidbtype *hwidb, list_header *l_hdr);
```

```
boolean idb_add_swidb_to_list(struct idbtype *edb, list_header *l_hdr);
```

6.6.3 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 `edb` 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 `edb` 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)
```

6.6.4 Iterate a List of Private IDBs Safely

To safely iterate over a private list of IDBs, call the `FOR_ALL_IDBS_IN_LIST_SAFE` macro. Use this macro to traverse a list that may change as it's being traversed and when speed of traversal is not a priority. This macro uses a safe iterator approach to protect against the list being changed while it's being traversed.

```
FOR_ALL_IDBS_IN_LIST_SAFE(list_header *list, iterator_t *ctx,
                           list_element *element, void *data)
```

Note The iterator is safely allocated and freed if the loop is allowed to run to completion. However, if a `break` or `return` is called within this loop, you MUST clean up the `iterator *context (ctx)` with an `iterator_delete(ctx)`.

Note The `FOR_ALL_IDBS_IN_LIST_SAFE` macro itself should not be called from interrupt level.

If a CLI accesses a private list of IDBs that is also used at interrupt level, call the `FOR_ALL_IDBS_IN_LIST_SAFE` macro. For example:

```
boolean
idb_is_present_on_list (list_header *l_hdr, const char *name)
{
    list_element *list_elt;
    void *idb;
    iterator_t *ctx;
    boolean found = FALSE;

    FOR_ALL_IDBS_IN_LIST_SAFE(l_hdr, &ctx, list_elt, idb) {
        if (!strcmp((idbtype *)idb->namestring, name)) {
            iterator_delete(ctx);
            found = TRUE;
            break;
        }
    }
    return (found);
}
```

6.6.5 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);
```

6.6.6 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);
```

6.7 IDB Data Structure Shrinking

The following information is taken from ENG-125646 and ENG-20143. Please refer to these documents for further details.

One aspect of the scalability effort in IOS software development is to shrink the core data structure requirements to reduce the memory cost for a large number of interfaces. Besides reducing memory requirements, there are a number of flow-on benefits from shrinking these data structures, including the following:

- With smaller data structures, the caching of data within first or second level caches can be more effective. With smaller data structures, there is a greater chance of data locality.
- Migrating the fields out from a common data structure to a feature-specific data structure greatly improves the software modularity of the system.
- With smaller core data structures, these data structures can be used in situations where it would have been prohibitive before, such as for distributed line cards.
- Routers can support a larger number of interfaces without having to increase the amount of physical memory required.

The primary effort for shrinking the IDB data structures involves migrating fields from the hardware and software IDBs to subblocks. Employing a more efficient scheme for storing Boolean flags in the IDBs is a secondary effort.

6.7.1 Dynamically Allocated Boolean Flags for IDBs

Throughout IOS, the `boolean` data type has been used to represent TRUE/FALSE (1/0) values. Currently, this data type is predefined as an `int`, a 32 bit value. Some time ago it was realized that a 32 bit object being used to store a single bit value could be wasteful, so a `tinybool` data type was defined, though the use of this seems to have been limited. Even so, there is a considerable waste of space in the central data structures due to the use of `boolean` and `tinybool`.

The following is a description of the alternative to using `booleans` and `tinybools` in the IDB structure. This is also be extended to replacing the currently statically allocated status bits in these structures.

Table 6-2 shows the overall cost of Booleans in the IDB structures.

Table 6-2 Boolean use in IDBs

Structure	booleans	tinybools	Total (bytes)
hwidb	20	44	124
swidb	37	60	208

Furthermore, the `hwidb` has 3 status words used for statically allocated bit flags.

Replacing these Booleans with a static bit array would save over 120 bytes in the `hwidb`, and nearly 200 bytes in the `swidb`.

Some of these Booleans are moved to subblocks, but often it is useful to have a flag on the IDB itself indicating whether the feature is enabled or not, or to flag conditions relating to the feature independent of the subblock.

However, like many of the fields with the `hwidb` and `swidb`, the Booleans are only relevant if the particular feature is enabled, or even included in the IOS image. Likewise, there are Booleans and fields that are only relevant for particular platforms. There has been considerable pressure against placing new fields in the IDB structures, so often what happens is that some status bits or values are reused if the feature writer knows that the owning feature does not run on this platform. Needless to say, this is poor programming practice, and can lead to obscure bugs and difficult-to-maintain code.

Similar to the subblock work, it is important that more modularity is achieved in the common structures, without losing the ability to access flags or Booleans; furthermore, only those flags relevant or referenced for the particular instance of IOS should be allocated or used. No platform or feature specific flags should be defined in common headers.

A static bit array is created in each data structure composed of an array of `uints`, each containing 32 bits of Boolean values, and these bits are dynamically allocated globally on a first-come, first served basis. Each request for a bit (Boolean) results in the allocation of a new index that can be used to set, clear or test the appropriate bit in the static bit arrays. Once allocated, the bit index is used for the same Boolean for all IDBs.

Since a lot of these flags and Booleans are platform and feature specific, it is not expected that the number of actual allocated bit indices match the existing number of Booleans and flags.

The Boolean Manager(BM) core support allows bit arrays to be placed on any IOS data structure by defining a new set of inline wrappers around some core inline functions, so other system data structures such as CDBs (Controller Descriptor Blocks) or `tty` lines can use the BM functionality; however, the scaling pressure for these data structures is not as critical.

The new APIs are `hwidb_bm_allocate()`, `hwidb_bm_assign()`, `hwidb_bm_clear()`, `hwidb_bm_set()`, `hwidb_bm_test()`, `hwidb_bm_allocate()`, `hwidb_bm_assign()`, `hwidb_bm_clear()`, `hwidb_bm_set()` and `hwidb_bm_test()`. Detailed descriptions of these APIs can be found in the *Interfaces and Drivers chapter of the IOS API document*.

The core BM code exists in `sys/h/bm_core.h` and `sys/if/bm_if.c`. Specific wrappers for the `swidb` and `hwidb` appear in `sys/h/bm_if.h`.

For the following examples, the wrappers for the `swidb` is used. Analogous wrappers exist for the `hwidb`. To allocate a bitmap index, the code is as follows:

```
#include "bm_if.h"
...
idb_bm_t my_bool_flag;
...
my_bool_flag = idb_bm_allocate(); /* Get bit index for my flag */
```

Bit index 0 is always reserved, so allocation starts from 1. The allocation is usually done at subsystem init time. The call always succeeds (a crashdump will occur otherwise) and returns an index that can be used in subsequent calls to test, set, clear or assign the actual bit values in the IDB, as such:

```
#include "bm_if.h"
...
idbtype *idb;
...
/* Check for my flag, returns TRUE/FALSE */
if (idb_bm_test(idb, my_bool_flag)) {
    ...
    idb_bm_clear(idb, my_bool_flag); /* Clear flag (assign FALSE) */
}
...
idb_bm_set(idb, my_bool_flag); /* Set flag (assign TRUE) */
...
boolean some_value;
...
/* Take TRUE/FALSE and assign it to flag */
idb_bm_assign(idb, my_bool_flag, some_value);
```

All of the functions assigning a value to the flag return void. For these functions, the index is checked, and if it is zero (i.e. usually an unallocated bit index) or larger than the allowed maximum, an `errmsg` with traceback is generated. Bit index 0 is always reserved, and always tests as 0, and no assignments are allowed to bit index 0.

For the bit test routine, if the bit index is zero, this is assumed that the subsystem has not actually allocated a bit number, which may mean that the subsystem has a lazy allocation policy, or that the subsystem doesn't even exist in this image. In this case, FALSE is always returned, as it is assumed that since no allocation is done, the flag has never been set, and ergo the flag must be FALSE (being the default value). This allows code that is not part of the subsystem to test flags with impunity, regardless of whether the allocating subsystem has actually requested a bit index (no error is generated). However attempting to *set* a bit using an unallocated bit index causes an error to be logged.

A similar situation exists as for the subblock index variable, in that any code that wishes to access a particular bit must have access to the variable containing the allocated bit index. This is not normally a problem, but in some circumstances code outside the main feature subsystem may wish to test or reference the bit, and if the subsystem containing the bit index variable is not present, there will be a link problem. For a detailed discussion of this, along with a range of suggested solutions, see ENG-20143.

6.8 IDB Helper Functions

Many functions fall into a class of functions that perform very common, but uncomplicated operations on IDBs.

6.8.1 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,
void *argument);

boolean idb_for_all_on_hwlist(idblist_t type, hwidbfunc_t function,
void *argument);
```

6.8.2 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);
```

6.9 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 Layer 3 or Layer 2 address, use the `idb_pak_vencap()` function.

```
boolean idb_pak_vencap(paktype *pak, long address);
```

6.10 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);
```

6.11 Getting and Setting IDB Fields

The `idb_hw_state_config()` function provides an opaque method for the update and retrieval of IDB members, allowing for member movement without requiring calling function changes. The `idb_hw_state_config()` function is a wrapper for the `*hwidb->state_config` function vector.

Note For information on the `*hwidb->state_config` function vector, refer to “HWIDB Control Function Vectors” in the *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.

```
boolean idb_hw_state_config(hwidbtype *hwidb, ulong opcode,
                            struct itemlist_ *list);
```

The `opcode` input parameter is used to specify the operation to be performed, such as retrieving or storing IDB configuration members; for example, these `opcode` settings are `IDB_CONTROL_READ_CONFIG` or `IDB_CONTROL_SET_CONFIG`.

The `list` parameter is used to specify the items to get or set in the state, configuration, or statistics fields in the IDB for a particular interface. The `itemlist_` structure contains an array of items to get/set.

```
itemdesc[n]           /* Descriptor for each item to be set/retrieved */
```

Each `itemdesc` in the array contains an indication of the item to be retrieved/modified and a memory location. This memory location is either data to be saved (`READ opcode`) or the location to store the current value (`SET opcode`).

The following functions are used when creating the `list` of items to get or set:

- 1 `item_desc_init()`—Updates an item list entry before passing the entry to the `idb_hw_state_config()` function.
- 2 `item_get_timestamp()`—Gets the timestamp from the item descriptor entry for later placement within the IDB structure.
- 3 `item_put_timestamp()`—Puts a timestamp from within the IDB structure into the item descriptor entry.
- 4 `item_list_malloc()`—Requests memory to pass to the `idb_hw_state_config()` function.
- 5 `itemlist_reset()`—Resets an item list, restoring the reference to the first entry.
- 6 `N_ITEM_LIST`—Defines a multiple entry item list to pass to the `idb_hw_state_config()` function.
- 7 `ONE_ITEM_LIST`—Defines a single entry item list to pass to the `idb_hw_state_config()` function.

See the reference pages for these functions and the `idb_hw_state_config()` function in the *Cisco IOS API Reference* for more information on these functions.

6.11.1 Defining a Single Item list Parameter

If the `list` contains a single item to get or set, the list can be placed directly into a static array using the `ONE_ITEM_LIST` macro:

```
ONE_ITEM_LIST(itemlist *name, ushort item, ushort length, ushort value)
```

For example, to use the macro `ONE_ITEM_LIST()` to propagate the setting from the command “[no] ignore dcd” to the `*hwidb->state_config` function vector, and from there to the `hwidb`, use the following code:

```
ONE_ITEM_LIST(counters, SERIAL_HW_IGNORE_DCD, sizeof(ulong), csb->sense);
if (!(hwidb, IDB_CONTROL_SET_CONFIG, &counters_list)) {
    /* errmsg(&msgsym(SOME_STATE_CONFIG_ERROR, INTERFACE_API)); */
    return (FALSE);
}
```

Note the relationship between the `ONE_ITEM_LIST` macro `name` argument (`counter`) and the `idb_hw_state_config()` function `item` argument (`&counters_list`).

6.11.2 Defining a Multiple Item list Parameter

If the `list` contains multiple items, there are two required (plus one optional) steps.

Step 1 You must first define the list and its array size via either of the following two functions shown here (the first creates a static array, the second a dynamic one):

```
N_ITEM_LIST(itemlist *name, ushort num)

void item_list_malloc (struct itemlist **item_list_param,
                      struct itemdesc **item_desc_param,
```

```
ulong item_count)
```

- Step 2** Next, you must initialize each one of these items as follows:

```
void item_desc_init (struct itemdesc_ *itemdesc,
                     ushort item,
                     ushort length)
```

- Step 3** Optionally, if you want to reuse the list created in Step 1, you must reset the list indices as follows:

```
void itemlist_reset (itemlist *ilist)
```

Also, reset the items using one of the following two formats. The first format is the safest; with the second format be sure to first research the `itemdesc_` structure fully.

```
void item_desc_init (struct itemdesc_ *itemdesc, ushort item,
                     ushort length);

itemdesc.handled = FALSE;
```

For example, this hypothetical code snippet will read two IDB error count “state” items, each of length `ulong`. It will then increment these numbers and write them back to the IDB:

```
itemlist *ether_list = NULL;
itemdesc *ether_item = NULL;

item_list_malloc(&ether_list, &ether_item, 2);
item_desc_init(&ether_item[0], ETHER_ONE_COLLISION, sizeof(ulong));
item_desc_init(&ether_item[1], ETHER_MULTI_COLLISION, sizeof(ulong));
if (!idb_hw_state_config(hwidb, IDB_CONTROL_READ_CONFIG, &ether_list)) {
    /* errmsg(&msgsym(SOME_STATE_CONFIG_ERROR, INTERFACE_API)); */
    free(ether_list);
    return (FALSE);
}
ether_item[0].u.l_value++;
ether_item[1].u.l_value++;
itemlist_reset(&ether_list);
ether_item[0].handled = FALSE;
ether_item[1].handled = FALSE;
if (!idb_hw_state_config(hwidb, IDB_CONTROL_SET_CONFIG, &ether_list)) {
    /* errmsg(&msgsym(SOME_STATE_CONFIG_ERROR, INTERFACE_API)); */
    free(ether_list);
    return (FALSE);
}
free(ether_list);
/*
 * printf ("New counts: %lu One collisions, %lu Multi-collisions",
 * ether_item[0].u.l_value,
 * ether_item[1].u.l_value);
*/
```

6.11.3 Timestamps Within the list Parameter

For timestamp manipulations, the `hwidb->state_config` function vector has two additional functions at its disposal. The setup by the calling routine is similar.

6.11.3.1 Retrieving a Timestamp from the IDB

To retrieve a timestamp within the IDB, the invoking routine must establish a local variable to house the timestamp, build the item list and call the `edb_hw_state_config()` wrapper, requesting a READ.

The `*hwidb->state_config` function vector must recognize the READ request type, the particular item to be retrieved, then copy it into the `list` array by calling the `item_get_timestamp()` function.

Step 1 The invoking routine must define the location of the timestamp.

```
sys_timestamp state_change_time;
ONE_ITEM_LIST(time, ITM_HW_STATE_TIME, sizeof(void *),
              state_change_time);
if (!edb_hw_state_config(hwidb, IDB_CONTROL_READ_STATE, &time_list)) {
    /* errmsg(&msgsym(SOME_STATE_CONFIG_ERROR, INTERFACE_API)); */
    return (FALSE);
}
/* printf ("Time of last state change was %x", *time_item[0].u.buffer);
 */


```

Step 2 The called `hwidb->state_config` function vector must copy the current timestamp into the location specified in the `list` array.

```
boolean my_read_state_fv (hwidbtype *hwidb, itemlist *ilist)
{
    itemdesc *itemdesc;
    while ((itemdesc = itemlist_getnext(ilist))) {
        . . . snip . . .
        if (itemdesc.item == ITM_HW_STATE_TIME) {
            item_get_timestamp(ilist, itemdesc, hwidb->state_time);
            itemdesc->length = sizeof(sys_timestamp);
            continue;
        }
    }
}


```

6.11.3.2 Saving a Timestamp into the IDB

To save a timestamp within the IDB, the invoking routine must establish the timestamp, build the item list and call the `edb_hw_state_config()` wrapper, requesting a SET.

The `*hwidb->state_config` function vector must recognize the SET request type, the particular item to be saved, then save it calling the `item_put_timestamp()` function.

Step 1 The invoking routine must define the location of the timestamp.

```
sys_timestamp state_change_time;

GET_TIMESTAMP(state_change_time);

ONE_ITEM_LIST(time, ITM_HW_STATE_TIME, sizeof(void *),
              state_change_time);
if (!edb_hw_state_config(hwidb, IDB_CONTROL_SET_STATE, &time_list)) {
    /* errmsg(&msgsym(SOME_STATE_CONFIG_ERROR, INTERFACE_API)); */
    return (FALSE);
}
/* printf ("Time of last state change set."); */


```

- Step 2** The called `hwidb->state_config` function vector must save the supplied timestamp within the `list` array.

```
boolean my_set_state_fv (hwidbtype *hwidb, itemlist *ilist)
{
    itemdesc *itemdesc;
    while ((itemdesc = itemlist_getnext(ilist))) {
        . . . snip . . .
        if itemdesc.item == ITEM_HW_STATE_TIME {
            hwidb->state_time = item_get_timestamp(ilist, itemdesc);
            continue;
        }
    }
}
```

6.11.4 Using the list Input Parameter

You must first invoke the `N_ITEM_LIST` or `ONE_ITEM_LIST` macro to generate the initialized `itemdesc` entries.

To populate these entries, call `item_desc_init()` specifying the particular entry number and the characteristics to include (the item, the `enum` value of the item and length, and the size of the item).

For example, the following code can be used for getting 3 list items from the Ethernet collision statistics for an interface and printing the results received.

```
N_ITEM_LIST(ether_col, 3);

item_desc_init(&ether_col_item[0], ETHER_HW_OUTPUT_ONE_COLLISION_STAT,
               sizeof(ulong));

item_desc_init(&ether_col_item[1], ETHER_HW_OUTPUT_TWO_COLLISION_STAT,
               sizeof(ulong));

item_desc_init(&ether_col_item[2], ETHER_HW_OUTPUT_THREE_COLLISION_STAT,
               sizeof(ulong));

if !(idb_hw_state_config(hwidb, IDB_CONTROL_READ_STATE, &ether_col_list))
{
    /* errmsg(&msgsym(SOME_STATE_CONFIG_ERROR, INTERFACE_API)); */
    return (FALSE);
} else {
/*
 * printf ("%lu one collision; %lu two collision; %lu three collision",
 *         ether_col_item[0].u.l.value,
 *         ether_col_item[1].u.l.value,
 *         ether_col_item[2].u.l.value )
 */
}
```

6.12 Protecting an IDB from the User

To protect an IDB from the user or, in other words, to disallow configuration and to hide an IDB from the user, follow the instructions in this section.

If you want some interfaces, such as backplane interfaces connecting various components in a distributed platform, to not be visible to the user or not configurable by the user, the following solutions are available:

- *Hidden and not configurable* by any mechanism (CLI or SNMP)
 - Follow this sequence: `idb_create()`, `idb_enqueue(hwidb)`, then `idb_unlink(hwidb)` when it is useful for the `hwidb->hw_if_index` IDB infrastructure, as part of `idb_enqueue()`, to remain intact and any code which references it, to continue to access the interface. For example, this method is usually the process followed by interface cards that are OIR'ed which are then expected to return to the system once the OIR operation is complete.
 - Remove the IDB from the system IDB list using `idb_unlink()`. Because the CLI parser traverses the system IDB list to find a related entry, parser lookup will fail.
- *Visible but not configurable* via CLI
 - To disallow configuration, for example of a backplane Ethernet interface, perform the following:
 - (a) Set the flag `hwidb->status |= IDB_NO_NVGEN`
 - (b) Register a routine in `reg_add_interface_command_ok()`.

The registry is a RETVAL (case service with a return value of boolean) and uses the `IDBTYP` as the index.

The argument passed is a pointer to CSB.

The registered routine will retrieve the `hwidb` pointer from `csb->interface` in case any further checking is needed from `csb->interface` to determine if the interface should be configurable.

Return `TRUE` to indicate processing may continue.

Return `FALSE` to indicate this interface should not be configurable.

Consider including a `printf()` error message in the function stating that the interface is not configurable and any additional help information. Since config commands can come from many sources, such a `printf()` should be conditional on `csb->resolvemethod` being equal to `RES_MANUAL`, indicating a human just typed the command.
- Not configurable by SNMP (interface must be visible)
 - To also disallow interface configuration by SNMP, you may use the `reg_add_ifmib_interface_configurable()` which provides a similar service as the `reg_add_interface_command_ok()` for CLIs to disallow interface configuration via SNMP.

6.13 The linktype enum

The new linktype enum was committed to 12.3 and 12.2S. It is a great improvement in the sync times over the old linktype enum.

The `LINK_TYPE` enum is a generic network layer type (IP, ARP, TAG, etc.). On arrival to the box, it is extracted from the packet. If it originates from the box, it has to be set. For example, when a packet arrives over 802.3, the type is extracted from the 802.2 SNAP type. The conversion to link type is done with the `type2link()` function.

The `LINK_TYPE` is saved in `pak->linktype`.

Here are some instructions when adding a new link type. These instructions are located at the end of the `linktype.enum` file in the `if` directory. (`if/linktype.enum`). The new link types should be declared *before* these instructions in that file.

Caution Please follow these instructions when adding a new link type. Because of In Service Software Upgrade (ISSU) considerations, link types *must* be consistent among branches.

- Step 1** Locate the latest mainline branch. When this was written people are actively developing in 12.0S, 12.1E, 12.2S, and 12.3T. The latest mainline at that point in time was 12.3 (georgia). Please use this logic accordingly in locating the latest mainline branch.
- Step 2** Choose the next available VALUE from the latest mainline branch and add your new link type names. The “LONG” name is what is returned by `link2name()`. The “SHORT” name is returned by `print_linktype()`.
- Step 3** Place this new link type after the last link type and BEFORE these instructions. These instructions and the link type data structures are in the file `if/linktype.enum`. Prevent conflicts with other developers by committing (after a code review by the component owners) this new value in the latest mainline branch.
- Step 4** Commit this value to your own branch. Do not worry if there are gaps in the numbering. `link2name()` and `print_linktype()` will handle this properly.

Note This process requires TWO commits.

Here are some examples of link types:

```
BEGIN LINKTYPE

ENUM link_t ENUM_NAME LINK

# invalid link type
ITEM LINK_ILLEGAL
    VALUE      0
    LONG      "ILLEGAL"
    SHORT     "illegal"
END ITEM

# IP ARP
ITEM LINK_ARP
    VALUE      1
    LONG      "ARP"
    SHORT     "arp"
END ITEM

# Reverse ARP
ITEM LINK_RARP
    VALUE      2
    LONG      "RARP"
    SHORT     "rarp"
END ITEM

# Xerox ARP
ITEM LINK_XARP
    VALUE      3
    LONG      "XARP"
    SHORT     "xarp"
END ITEM

# HP's 802 version of ARP
ITEM LINK_PROBE
```

nextsub Subinterface List (*new in 12.2T*)

```

VALUE      4
LONG      "PROBE"
SHORT     "probe"
END ITEM

```

6.14 nextsub Subinterface List (*new in 12.2T*)

The following text is from CSCdu44151. Please refer to this DDTs for more complete information on the `nextsub` subinterface list feature.

Subinterfaces are chained via a singly-linked list on the software IDB using the `nextsub` pointer. Elements are added to this list in sorted order based on the subinterface number (`sub_number`). This is done by walking the `nextsub` list and comparing the `sub_number` to find the correct location. When elements are removed from this list, the list is scanned looking for the previous element so that it may be linked to the next element.

It is easy to see that this scheme does not scale well when you consider that 30,000 subinterfaces may be on that list. This is the case if we are terminating 30,000 PPPoX sessions on one box.

Virtual-access subinterfaces are added and removed from the `nextsub` list during call setup and teardown. Whatever data structure is used to replace the singly-linked list, it must be able to handle quick insertions and deletions. This capability should be considered the most important design requirement. The per-session memory requirements should be taken into consideration as well.

To maintain current functionality, the list of subinterfaces needs to be walked in sorted order. This order is needed by user-interface code (and possibly other code).

Another requirement that should be kept in mind is for there to be a mechanism by which a specific subinterface may be located (or not located) in a reasonable amount of time. This capability is needed by the parser (see `find_swidb_from_hwidb()`) to locate a specific subinterface.

One possible solution was to implement the subinterface list as a Red-Black (RB) tree as described in the IOS Programmer's Guide:

http://wwwin-enged.cisco.com/ios/doc/pg/12_1/utrees.html#307

One concern with using an RB tree is that the insertion/deletion time isn't O(1), and there may be additional time needed to keep the tree balanced. The balancing would likely occur during insertions/deletions when we can afford it the least.

Use the following functions to manipulate subinterfaces:

To add one `hwidb` and all of its associated `swidbs` to the `hwidbQ` and `swidbList`, call the `edb_enqueue_full()` function.

```
void idb_enqueue_full (hwidbtype *hwidb);
```

To remove a subinterface IDB from `swidbList` and the `hwidb->firstsw->nextsub` queue, call the `edb_erase_subif()` function.

```
void idb_erase_subif (edbtype *swidb, boolean verbose);
```

To verify that only one `swidb` exists for this `hwidb` (that is, no subinterfaces), call the `edb_hw_get_only_swidb()` function.

```
edbtype* idb_hw_get_only_swidb(hwidbtype* hwidb);
```

To initialize/reinitialize a subblock IDB, call the `idb_subif_init()` function.

```
boolean idb_subif_init(idbtype *idb,
                      idbtype *subidb,
                      ulong if_number,
                      ulong subidbnum);
```

To add the hwidb and master swidb to the global queue, call the `make_idb_visible()` function.

```
void make_idb_visible (hwidbtype *hwidb);
```

6.15 Interface Locking Mechanism (*new in 12.1*)

This section describes the interface locking mechanism that has been implemented to provide a common method to prevent Cisco IOS process contention when run-to-completion assumptions are not preserved because of non-blocking IPC. The information for this section comes from EDCS-197002.

6.15.1 Introduction

Cisco IOS was designed using the run-to-completion assumption. This model works well in a single processor environment with blocking IPC, but in a multiprocessor environment with non-blocking IPC, the model can cause multiple processes to contend for the same resource. These process contentions can lead to redundant commands being invoked to service the same event. In some cases, two separate processes using the same data can lead to corruption of data structures. For example, process A may try to tear down all ATM VCs in response to a line alarm. After running for a while, process A decides to give up the CPU. Then process B comes along and also wants to do something with the VCs on that interface. If it does something like freeing memory that process A depends on, then process A crashes when it runs next.

Typically, the approach used to resolve this class of problems has been to fix each occurrence one by one. This approach does not scale since it must be repeated for each interface and platform. It also does not provide a common infrastructure that could be leveraged across Cisco IOS. The interface locking mechanism provides a general solution.

6.15.2 Design

A HWIDB level semaphore (within a HW subblock) is introduced that is required to be taken before any application layer code makes a call that accesses the interface. Interface lock/unlock calls are added whenever accessing an interface. Inside the lock/unlock calls, checks are made to see whether the feature is supported by the interface type (for example, ATM). If the feature is not supported, the call is executed in the same manner as it is currently without any use of the semaphore. A simple function is provided to allow driver code to detect and report function calls without the use of the semaphore.

6.15.3 Example of Interface Locking

When processing multiple commands for a given interface, the application code should lock the interface outside the main loop to ensure that other processes do not use the interface until the process is done. In this example, the process waits until the lock is obtained before proceeding.

For example:

```
if (if_lock_get(hwidb)) {
    FOR_ALL_SWIDBS_ON_HW() {
        ... do something ...
    }
    if_lock_release(hwidb);
}
```

6.15.4 Example of Using Interface Locking to Service Unblocked Interfaces

The following example shows how application layer code could take advantage of the locking feature to service non-blocked interfaces while waiting for a blocked interface to become available.

```
FOR_ALL_HWIDBS(hwidb) {
    if (if_lock_get_1try(hwidb)) {
        ... use locked interface ...
        if_lock_release(hwidb);
    } else {
        enqueue(busy_q, hwidb);
    }
}
while ((hwidb = dequeue(busy_q))) {
    if (if_lock_get_1try(hwidb)) {
        ... use locked interface ...
        if_lock_release(hwidb);
    } else {
        enqueue(busy_q, hwidb);
    }
}
```

6.15.5 Detection of Interface Access without Lock

To provide a simple way to detect cases when interface driver code is called without using the locking logic, the `if_lock_verify()` function is provided that may be inserted in interface driver functions. If the check finds that the driver code has been called without locking the semaphore, an error message and traceback are printed to identify the offending application code.

The function returns TRUE if the interface is locked properly or if HWIDB locking is not supported on the interface. It returns FALSE if HWIDB locking is supported on the interface and the semaphore is not locked.

```
/*
 * if_lock_verify
 *
 * Verify that interface lock is being used properly.
 */
boolean if_lock_verify (hwidbtype *hwidb)
{
    if_lock_sb_t *if_lock = idb_get_hwsb_inline(hwidb, HWIDB_SB_IF_LOCK);

    if (if_lock) {
        if (if_lock->sem->s) {
            pid_t this_pid;

            process_get_pid(&this_pid);
            if (if_lock->sem->owner == this_pid) {
                return (TRUE);
            }
        }
    }
}
```

```

        } else {
            errmsg(&msgsym(BADLOCK, IF), hwidb->hw_namestring,
                   "not owner");
        }
    } else {
        errmsg(&msgsym(BADLOCK, IF), hwidb->hw_namestring,
               "not locked");
    }
    return (FALSE);
}
return (TRUE);
}

```

Any driver that is adding support for interface locking should add calls to verify proper interface lock usage wherever appropriate in the interface driver code.

6.15.6 Data Structures

A new HW subblock is added for interfaces that choose to enable the interface locking support.

```

typedef struct if_lock_sb_t_ {
    HWSB_BASE; /* Common subblock header */
    watched_semaphore *sem;
    ulong             nesting;
    ulong             timeout;
} if_lock_sb_t;

```

Definitions (bitmasks) are added to indicate which interface types are capable of supporting interface locking.

```

#define IDB1_LOCKABLE (IDB_ATM)
#define IDB2_LOCKABLE 0x00000000
#define IDB3_LOCKABLE 0x00000000

```

6.15.7 End User Interface

A new error message is introduced to report cases where device driver code is called without proper use of the semaphore. The error message also generates a traceback to make it simple to fix such problems.

6.15.8 Adding Interface Locking Support on an Interface

When a device driver chooses to enable interface locking support, calls to `if_lock_verify()` should be added in the driver code wherever application code interfaces with the driver. This provides a quick way to catch and fix invalid lock uses. Also, the driver needs to ensure that the application code for the given interface type supports interface locking. This can be verified by checking which interface types are enabled in the `IDB*_LOCKABLE` defines.

6.15.9 Steps to Enable Interface Locking Support

To enable interface locking support, follow these steps:

- Step 1** Verify that the interface type supports interface locking (see `IDB[123]_LOCKABLE` defines).

Step 2 Add a call to `if_lock_enable()` in the device driver `init` routine.

Step 3 Add calls to `if_lock_verify()` wherever application code calls the driver.

Test the code and look for and fix any occurrences of the invalid lock error messages. (Add locks in the application layer code as required.)

6.16 The Dev-Object Model

The Dev-Object Model is used for cross-OS and cross-platform driver sharing. The model provides a common device driver structure or reusable device “plugin” for software developers to use when developing device drivers for their platform. This model allows you to avoid repeating the cost of device driver development for each driver written and to avoid any maintenance, updates, or feature upgrades for the device drivers for each device driver implementation, as well as the testing for verification of operation. Also, multiple device drivers for the same device type are no longer needed and no longer compiled into the same software image when using the Dev-Object Model, which will decrease the size of the image, and increase the available memory space.

Each dev object-based driver implements the dev object common function vector table, which allows the upper-layer code to do the device initialization, device enable, device destroy, and so on. The common device driver object for the Dev-Object Model is the `dev_object_t` structure for the common function vector table that is used to control device objects. This table contains a minimal set of information (data and functions) that is common to most all devices; basic device function vectors, a device base address, a device revision number, a device bit flag, and the current state of the device.

Devices provide additional functionality by publishing a device-specific (or device class) function vector table that gives the upper-layer code knobs to control that specific device (or class of device). For example, a SONET framer device would use a function vector table that allows you to set overhead bytes, set clocking source, collect statistics, and so on.

Another set of vectors is defined that allows devices to access OS provided services, such as `printf()`, `malloc()`, `list()`, and `queue()` routines. These vectors are initialized once by the OS that supports dev objects, and then any dev object-based driver can access those services. These are the vectors that come primarily from standard libraries.

Support is also provided for the Interrupt Manager (this is really an example of how the Dev-Object Model can be extended to support new features, rather than being part of the base model). In this case, a common header file is defined that publishes the related APIs, then dev object-based drivers can make use of this new functionality by including that header file. In this case, there is also some OS-side support that is also required.

The standard for representing the various data types in the Dev-Object Model is the Portable Operating System Interface (POSIX) standard from the IEEE (P1003.1, Draft 7, June 2001, Open Group Technical Standard Issue 6). The Dev-Object Model has been adopted across multiple linecards and shared port adapters on IOS, IOX, QNX, IOU, and diags (for example, on the 7300, 7600, 10000, 12000 platforms).

6.16.1 Interacting with Dev Objects

Any device that will be used on more than one hardware platform is well suited to use the Dev-Object Model. Many drivers for chips on SIP linecards and the SPA (Shared Port Adapter) driver make use of the model. For example, SPA driver interactions include:

- The SPA common code, such as the OIR SPA common code, can invoke common Dev-Object Model function vectors.

- The SPA-specific driver invokes Dev-Object Model function vectors via the Interface Mapper. The Interface Mapper provides the glue between the SPA i/f driver and the Dev-Object, and keeps the SPA i/f driver independent from the device specifics.

Dev objects are controlled in the Dev-Object Model as follows:

- Common code uses the `dev_object_fvt` vectors (for example, `dev_attach`, `dev_init`, `dev_oper_enable`).
- The Dev-Object Model macros are provided for consistency and ease-of-use (for example, `DEV_INIT(spa_plugin -> my_device)`).
- The device state is maintained for sanity checking.
- The Dev-Object Model includes a common set of debug flags (enabled/disabled via the **debug hw-module** CLI command).

When the dev object interacts with a host, the device name (`dev_location`) gives unique context of where the device fits in the system; for example, “ATM4/0 SAR”. Errors are reported via the `dev_error_report` call. The dev object builds the error string before invoking the call that leverages platform error reporting facilities.

When porting dev objects to another OS, the dev objects are simply copied over (eventually will be single-sourced). A script is provided to verify compliance to Dev-Object Model (currently `~donwall/bin/devchk`). The upper-layer driver is provided (the i/f Mapper and SPA driver are ported).

6.16.2 Introduction to Using the Dev-Object Model

This subsection provides a brief introduction on how to develop a device driver using the Dev-Object Model. You will need to see EDCS-188171 for detailed information on the Dev-Object Model and how to develop a specific device driver using the Dev-Object Model.

6.16.2.1 The Dev-Object Model Code Directory Structure

The `dev_object_t` structure and associated data are placed in a “`dev_object.h`” header file and are committed into `/vob/cisco.comp/devobj/include` on single-source branches (before single-source, it was committed into the `/vob/ios/sys/chips/common` directory).

For every specific device driver developed with the common dev object approach, you create all of the files necessary for a given device. At a minimum, this includes; a `dev_device_name.c` and a `dev_device_name.h` file. All specific device drivers developed with the common device object are placed under the `vob/ios/sys/chips/device_name` directory. This provides a location that will begin to form a library of device drivers for many device types and allows other software developers to find the one that they are looking for, otherwise, write the specific device driver for use by themselves and anyone else who may have a future need for that device. Thus, only one device driver for any given device type is developed and used by all of the software groups, whether Diagnostics, IOS, CatOS, ION, or some other new group.

The Dev-Object Model code directory structure is as follows:

- `sys/chips/common` for the dev object common code.
- `sys/chips/os` for the OS dependent support for dev objects.
- `sys/chips/common/ios` for the IOS-specific support for dev objects
- `sys/chips/some_device` for the driver code for a specific device object

6.16.2.2 Rules for “C” #include Directives

Some platforms don’t allow relative include paths. Consequently, include directives such as the following will not compile on some platforms:

```
#include "../common/dev_master.h"
```

The proper way to specify such an include is as follows:

```
#include <dev_master.h>
```

Thus, each platform puts the `chips/common/` directory in the “make” include path name space, and device drivers do not use relative includes.

For single-source branches, the proper way to specify such an include is as follows:

```
#include COMP_INC(devobj, dev_master.h)
```

6.16.2.3 Dev-Object Model API

The Dev-Object Model API is in `dev_list.h`, `dev_queue.h`, `dev_master.h`, `dev_buginf.h`, `dev_mutex.h`, `dev_util.h`, `dev_object.h`, `dev_types.h`, and `dev_macros.h`.

When writing dev object-based driver, only `dev_master.h` needs to be included in the driver to pick up all the required API definitions.

Note that the `dev_util.h` file should not be included by dev objects themselves. The `dev_util.h` file is only used by the OS-specific code when initializing the OS provided vectors. This file provides the following list of all the OS-provided API functions that are available for use by dev object-based drivers:

```
typedef struct dev_os_func_cfg_t_ {
    dev_printf_t           printf;
    dev_sprintf_t          sprintf;
    dev_snprintf_t         snprintf;
    dev_buginf_t           buginf;
    dev_printf_t           crash_printf;
    dev_memcpy_t           memcpy;
    dev_memcmp_t           memcmp;
    dev_memset_t           memset;
    dev_strcpy_t           strcpy;
    dev_strncpy_t          strncpy;
    dev_strncat_t          strncat;
    dev_malloc_t            malloc;
    dev_malloc_aligned_t   malloc_aligned;
    dev_free_t              free;
    dev_qsort_t             qsort;
    dev_bsearch_t           bsearch;
    dev_delay_t             delay;
    dev_delay_t             nano_delay;
    dev_delay_t             msec_delay;
    dev_read32_t            read32;
    dev_write32_t           write32;
    dev_and32_t             and32;
    dev_or32_t              or32;
    dev_set32_t             set32;
    dev_read16_t            read16;
    dev_write16_t           write16;
    dev_and16_t             and16;
    dev_or16_t              or16;
    dev_set16_t             set16;
```

```

dev_read8_t           read8;
dev_write8_t          write8;
dev_and8_t            and8;
dev_or8_t             or8;
dev_set8_t            set8;
dev_checkqueue_t      checkqueue;
dev_dequeue_t          dequeue;
dev_p_dequeue_t       p_dequeue;
dev_enqueue_t          enqueue;
dev_p_enqueue_t       p_enqueue;
dev_p_requeue_t       p_requeue;
dev_p_swapqueue_t     p_swapqueue;
dev_p_unqueue_t       p_unqueue;
dev_p_unqueueuenext_t p_unqueueuenext;
dev_queue_init_t       queue_init;
dev_requeue_t          requeue;
dev_unqueue_t          unqueue;
dev_insqueue_t         insqueue;
dev_remqueue_t         remqueue;
dev_queryqueuedepth_t queryqueuedepth;
dev_peekqueuehead_t   peekqueuehead;
dev_validqueue_t       validqueue;
dev_list_init_t        list_init;
dev_list_memory_init_t list_memory_init;
dev_list_move_t         list_move;
dev_list_insert_element_t list_insert_element;
dev_list_remove_element_t list_remove_element;
dev_list_create_t       list_create;
dev_list_destroy_t     list_destroy;
dev_list_set_automatic_t list_set_automatic;
dev_list_set_interrupt_safe_t list_set_interrupt_safe;
dev_list_set_info_t    list_set_info;
dev_list_get_info_t    list_get_info;
dev_list_set_action_t  list_set_action;
dev_list_get_action_t  list_get_action;
dev_list_excise_t      list_excise;
dev_list_is_automatic_t list_is_automatic;
dev_list_check_t        list_check;
dev_get_timestamp_t    get_timestamp;
dev_elapsed_time_t     elapsed_time;
dev_elapsed_time64_t   elapsed_time64;
dev_mutex_init_t        mutex_init;
dev_mutex_destroy_t    mutex_destroy;
dev_mutex_lock_t        mutex_lock;
dev_mutex_unlock_t     mutex_unlock;
dev_mutex_trylock_t    mutex_trylock;
} dev_os_func_cfg_t;

```

Here are the details for each dev object API definition:

```

int32_t  dev_printf(const char *, ...);
int32_t  dev_sprintf(char *, const char *, ...);
int32_t  dev_snprintf(char *, size_t n, const char *, ...);
int32_t  dev_buginf(const char *, ...);

void *   dev_memcpy(void *, const void *, size_t);
int32_t  dev_memcmp(const void *, const void *, size_t);
void *   dev_memset(void *, int, size_t);
char *   dev_strcpy(char *dest, const char *src);

```

```

char *      dev_strncpy(char *dest, const char *src, size_t n);
char *      dev_strncat(char *dest, const char *src, size_t n);
void *      dev_malloc(size_t);
void *      dev_malloc_aligned(uint32_t, uint32_t);
void        dev_free(void *);
void        dev_qsort(void *base, size_t nel, size_t size,
                     int (*compare) (const void *, const void *));
void *      dev_bsearch(const void *key, const void *base,
                       size_t nel, size_t size,
                       int (*compare)(const void *, const void *));

/*
 * dev_delay() is available to devices that need a fixed time-delay
 * mechanism between device operations. One microsecond or one
 * nanosecond is the amount of time delay per unit.
 */
void        dev_delay(uint32_t);

/*
 * Platform supplied register access routines.
 *
 * read - return register
 * write - register = value
 * and - register = register & value
 * or - register = register | value
 * set - register = ((register & ~mask) | (val & mask))
 *
 * The set function will set all bits that are '1' in mask to the
 * corresponding bit value of the val parameter. For example, if
 * mask = 0x00000011 and val = 0x00000011 then bit 0 and 4 of register
 * will be set to 1. If mask = 0x00000011 and val = 0x00000001 then
 * bit 0 and 4 of register will be set to 1 and 0 respectively. Bit
 * positions not specified in the mask are not changed.
 *
 * Due to inconsistencies in platform bus accesses special routines
 * must be defined.
 */
uint64_t    dev_read64(volatile uint64_t *);
void        dev_write64(volatile uint64_t *, uint64_t);
void        dev_and64(volatile uint64_t *, uint64_t);
void        dev_or64(volatile uint64_t *, uint64_t);
void        dev_set64(volatile uint64_t *, uint64_t mask, uint64_t val);

uint32_t    dev_read32(volatile uint32_t *);
void        dev_write32(volatile uint32_t *, uint32_t);
void        dev_and32(volatile uint32_t *, uint32_t);
void        dev_or32(volatile uint32_t *, uint32_t);
void        dev_set32(volatile uint32_t *, uint32_t mask, uint32_t val);

uint16_t    dev_read16(volatile uint16_t *);
void        dev_write16(volatile uint16_t *, uint16_t);
void        dev_and16(volatile uint16_t *, uint16_t);
void        dev_or16(volatile uint16_t *, uint16_t);
void        dev_set16(volatile uint16_t *, uint16_t mask, uint16_t val);

uint8_t     dev_read8(volatile uint8_t *);
void        dev_write8(volatile uint8_t *, uint8_t);
void        dev_and8(volatile uint8_t *, uint8_t);
void        dev_or8(volatile uint8_t *, uint8_t);
void        dev_set8(volatile uint8_t *, uint8_t mask, uint8_t val);

```

```

/*
 * Register access routines for devices requiring extended addressing,
 * i.e. more than 32 bits. In order to support this for platforms
 * using 32 bit pointers, the address parameter is not a pointer but
 * a 64 bit integer.
 */
uint64_t dev_extadr_read64(uint64_t);
void dev_extadr_write64(uint64_t, uint64_t);
void dev_extadr_and64(uint64_t, uint64_t);
void dev_extadr_or64(uint64_t, uint64_t);
void dev_extadr_set64(uint64_t, uint64_t mask, uint64_t val);

uint32_t dev_extadr_read32(uint64_t);
void dev_extadr_write32(uint64_t, uint32_t);
void dev_extadr_and32(uint64_t, uint32_t);
void dev_extadr_or32(uint64_t, uint32_t);
void dev_extadr_set32(uint64_t, uint32_t mask, uint32_t val);

uint16_t dev_extadr_read16(uint64_t);
void dev_extadr_write16(uint64_t, uint16_t);
void dev_extadr_and16(uint64_t, uint16_t);
void dev_extadr_or16(uint64_t, uint16_t);
void dev_extadr_set16(uint64_t, uint16_t mask, uint16_t val);

uint8_t dev_extadr_read8(uint64_t);
void dev_extadr_write8(uint64_t, uint8_t);
void dev_extadr_and8(uint64_t, uint8_t);
void dev_extadr_or8(uint64_t, uint8_t);
void dev_extadr_set8(uint64_t, uint8_t mask, uint8_t val);

/*
 * timestamp related functions
 */
void dev_get_timestamp(volatile sys_timestamp *);
/*
 * elapsed time functions return the number of msecs that have
 * passed since the timestamp was taken.
 */
uint32_t dev_elapsed_time(volatile sys_timestamp *);
uint64_t dev_elapsed_time64(volatile sys_timestamp *);

dev_status_mutex_t dev_mutex_init(dev_mutex_t *mutex);
dev_status_mutex_t dev_mutex_destroy(dev_mutex_t *mutex);
dev_status_mutex_t dev_mutex_lock(dev_mutex_t *mutex);
dev_status_mutex_t dev_mutex_unlock(dev_mutex_t *mutex);
dev_status_mutex_t dev_mutex_trylock(dev_mutex_t *mutex);

```

The Read/Write macros in `dev_macros.h` allow the platform to use either direct or indirect access to device registers. The indirect access can avoid replicated function pointers per dev object (for example, IOX). The 8/16/32 bit register access macros allow the individual device object to use the appropriate macros for the hardware.

The Dev-Object Model API includes the following macros for platforms to access device registers:

- 1** `DEV_READ32`
- 2** `DEV_READ16`

Default Box-Wide IEEE MAC Address

```

3  DEV_READ8
4  DEV_WRITE32
5  DEV_WRITE16
6  DEV_WRITE8
7  DEV_AND32
8  DEV_AND16
9  DEV_AND8
10 DEV_OR32
11 DEV_OR16
12 DEV_OR8
13 DEV_SET32
14 DEV_SET16
15 DEV_SET8

```

These macros are described in detail in the *Cisco IOS API Reference*.

6.17 Default Box-Wide IEEE MAC Address

The `default_mac_addr[]` global variable represents the default box-wide IEEE MAC address and is constructed in one of the following ways during system initialization:

- From the platform-supplied value on a few platforms (for example, c10k).
- From the first Ethernet interface MAC address.
- From the first IEEE interface MAC address if the network device does not host any Ethernet interfaces.
- From the MAC address, which is constructed from the timestamp and the serial number of the device/platform.

Once constructed, this default MAC address value is set on all non-IEEE interfaces that inherently do not have MAC addresses.

The `default_mac_addr[]` usages are:

- It is used whenever there is a need to have a MAC address and the interface type does not have one such as ATM/Serial interfaces or vtemplate HWIDBs, which do not support IEEE 802 MAC addresses.

It is also used to create a probable unique number via `create_unique_number()` function.

Platform-Specific Support

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.

Note Cisco IOS platform-specific support development questions can be directed to the ios_platforms@cisco.com alias.

This chapter includes the following topics:

- Platform-Specific Initialization: Overview
- main() -> Fundamental Initialization
- main() -> Memory Initialization
- main() -> Exception Initialization
- main() -> Console Initialization
- init_process() -> Packet Buffer Initialization
- init_process() -> Non-volatile Storage Initialization
- init_process() -> Interface Initialization
- init_process() -> Filesystem Initialization
- init_process() -> Terminal Line Initialization
- init_process() -> Configuration Verification Initialization
- init_process() Final Tasks
- Platform-Specific Strings
- Platform-Specific Values
- Interface ID Format
- How to Develop New Platform Support

List of Platform Code Terms

Code is either platform-independent or platform-dependent:

- 1 Platform-independent Code

2 Platform-dependent Code—There are two types of platform-dependent code:

- (a) Platform-specific Code
- (b) Platform-generic Code

These platform code terms are listed below in alphabetical order:

Platform-dependent Code

Platform-dependent code is compiled for a specific platform. Platform-dependent code contains instructions, idioms, etc., that are specific for a given platform, and thus, the compiled binary works only for that platform. For example, platform-specific code and platform-generic code is platform-dependent code.

Platform-generic Code

Platform-generic code uses semantics that are specific to the platform. However, it is *not* obvious upon inspection that such code is designed to execute on a specific platform. Platform-generic code depends exclusively upon conditional compilation paths to produce binaries that are specific to a given platform. As such, platform-generic code is a type of “template” where algorithms remain the same but the data manipulated differs. These platform-generic “templates” should be considered as part of the platform-generic API shared by all platforms. For example, platform-generic code is used for the same purpose by multiple platforms, but is customized per platform by separate platform-specific `cisco_platform.h` (see the `machine` directory) header files and/or by conditional compilation, and the resulting object code is platform-specific. For example, `interface.c` is platform-generic code. Also, the `ipfast_flow_les.c`, `ipfast_frag.c`, and `isdn.c` files are platform-generic.

Platform-independent Code

Typically, platform-independent code is independent of both the CPU and RTE (run-time environment). However, the generated binaries are *always* CPU-specific. Examples of platform-independent code include routing protocols, the scheduler, the error logger, and other high-level features. For instance, the `main()` function is platform-independent.

Platform-specific Code

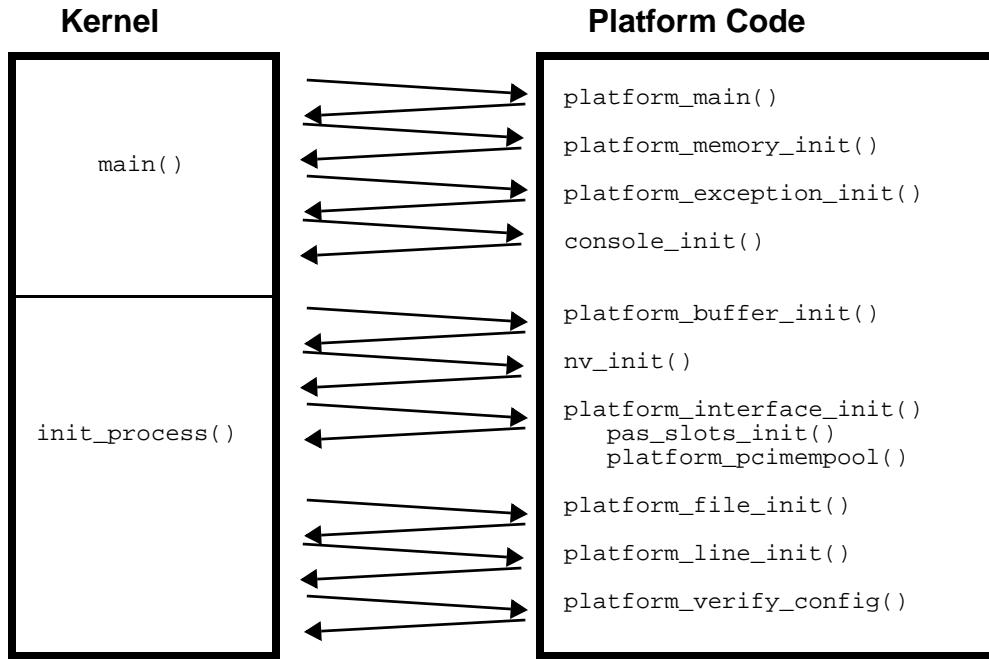
Platform-specific code uses semantics that are specific to the platform. It is obvious upon inspection that platform-specific code is designed for a given platform. Typically, files and/or directories have the platform name in the file name. Examples of platform-specific code are the bootstrap code, device drivers, and Flash memory drivers. Also, typically included in platform-specific 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. The initialization of various hardware devices, such as timers, interrupt controllers, and port adapters, is also performed by platform-specific code. For example, the `platform_memory_init()` function is platform-specific.

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, interrupt controllers, and port adapters, 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. Figure 7-1 shows the initialization hooks that are supplied by the system code.

Figure 7-1 Developer Hooks for Platform Support



The first platform-specific functions are called by `main()` after the system has loaded an image and started running. These platform-specific functions are called almost immediately after the system starts executing, after the platform's CPU type has been defined, and before the scheduler initialization.

For example, the platform's CPU type and family are first defined as follows:

```
main()
{
    cpu_type    = platform_get_value(PLATFORM_VALUE_CPU_TYPE)
    cpu_family = platform_get_value(PLATFORM_VALUE_FAMILY_TYPE)
    .
    .
    .
    ... main() continues ...
}
```

Note Platform-specific file names have the following format: `platform_xxxx.cxxxx_console.c`, and so on, where `xxxx` identifies the specific platform.

Next, the `main()` platform-specific functions are called in the following order:

- 1 `platform_main()` performs basic initialization actions, such as parsing cookie PROMs and turning on run LEDs.
- 2 `platform_memory_init()` declares memory regions and pools to the system code using the functions supplied by the memory management code.
- 3 `platform_exception_init()` initializes exception and interrupt handlers.
- 4 `console_init()` initializes the system console.

The last task `main()` has is to initialize and start the scheduler. At this point, `init_process()` has already been created and is ready to run. When `init_process()` runs, it completes all Cisco IOS core initialization tasks, although additional processes will have to run to completely initialize an IOS Network Device.

The remaining functions are called later during `init_process()` initialization. The remaining functions are called in the following order:

- 1 `platform_buffer_init()` initializes buffers with buffer sizes from 100 bytes to 18 Kbytes.
- 2 `nv_init()` initializes non volatile storage.
`platform_nvvar_support()` indicates the NVRAM variable support.
- 3 `platform_interface_init()` initializes network interfaces. This platform-specific routine may also call the following platform-specific routines:
 - `pas_slots_init()` initializes port adapters.
 - `platform_pcimempool()` allocates the platform device's memory space.
- 4 `platform_file_init()` sets up the platform's filesystem, including the device's non volatile storage.
- 5 `platform_line_init()` initializes terminal lines (`tty` or `vty`) other than the console.
- 6 `platform_verify_config()` checks that the hardware configuration and respective hardware levels matches the software configuration and respective software levels.

There is much more involved in the `init_process()` work other than the above platform-specific functions. For example:

- All subsystems are initialized.
- All network interfaces are initialized, including the chip and the software structures.
- Other components, such as hardware/firmware ASICs which assist in packet forwarding and switching, are also initialized.

The above platform-specific functions are described in more detail in the following sections.

7.2 main() -> Fundamental Initialization

After the platform's CPU type has been defined, `platform_main()` is called by `main()`. The `platform_main()` function allows platforms to perform basic initialization actions.

7.2.1 platform_main() Entry Conditions

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.

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. Therefore, you must take great care to do the absolute minimum in this function.

7.2.2 platform_main() Tasks

Although each platform will perform different tasks, some common examples of processing are:

- Disable console interrupts until system components have been set up to enable such interrupts.

- Save “reason last reset” as saved previously by the ROM monitor.
- Reset system-wide control registers.
- Reset independent interfaces and clear any pending interrupts from them.
- Parse the cookie PROMs.

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 non modular platforms). The cookie is mapped into the platform’s memory address space and is readable by the system code.

- Establish the end-of-scheduler routine.

This platform-specific routine is called at the bottom of the scheduler loop and is commonly used to “unthrottle” interfaces. Throttling occurs when the network device is overcommitted and can no longer efficiently process packets. Network interfaces are prohibited from receiving packets and can only transmit those already in the system. Because of the design of the scheduler loop, reaching the end indicates that the network device is no longer overloaded. At this point it is reasonable to reenable packet receipt from network interfaces.

- Turn on the system LED, if equipped. The particular LED and color setting depends on the platform, for example:
 - The Cat 5810 “Aries” will set the status LED to RED (STOPPED).
 - The Cisco AS5400 Universal Gateway will set the “system running” LED.

Other tasks are performed within [platform_main\(\)](#) processing. Remember to try and keep this routine to a minimum. Many of these could and probably should be done in later routines:

- Cat 5810 “Aries” will store the global maximum number of ATM interfaces on this network device.
- Cisco 12000 GSR BFRP will set the system state to “NOT YET DETERMINED”.
- Cisco 12000 GSR BFRP will obtain the hardware version and revision levels.
- Cisco uBR920 - Lowend Universal Broadband Router will reset and initialize the dependent components.
- Cisco uBR920 - Lowend Universal Broadband Router will setup the default MAC address.
- Cisco AS5400 Universal Gateway will retrieve startup and io-cache flags from NVRAM.
- Cisco AS5400 Universal Gateway will set flow-thru DRAM cache as supported by the motherboard.
- Cisco AS5400 Universal Gateway will reset all interfaces and clear any pending interrupts.

For example, the [platform_main\(\)](#) platform-specific function can be as follows:

```
... main() continued ...
platform_main()          platform_xxxx.c
/* for example:
my_cookie =                      specifies the hardware ID
my_revision =                     specifies the hardware revision level
my_rev_string =                   specifies the hardware revision string
my_reset_reason =                 specifies the last time the system went down
set_led(LED_LOCATION, COLOR_GREEN)  specifies the platform's front panel LED's
setting
kernel_set_sched_loopback(my_specific_routine)  executes a platform-specific
function. For example, a routine that collects statistics and puts them in buckets to be picked up later.
```

```

.
.
.
... main() continues ...

```

7.2.3 Examples: Fundamental Initialization

The following examples show how to implement [platform_main\(\)](#).

7.2.3.1 ubr7200 platform_main()

The following example shows how the Cisco ubr7200 Universal Band Router uses [platform_main\(\)](#) to perform fundamental initialization. The Cisco ubr7200 Universal Band Router [platform_main\(\)](#) routine is found in the `src-4k-cr7200/platform_cr7200.c` source file.

This hook allows the ubr7200 to:

- 1 Save the reason for last reset.
- 2 Build platform-dependent strings.
- 3 Set the end-of-scheduler routine. This routine will periodically trigger network interfaces “reawakening” if they were previously prohibited from receiving packets (“throttled”) because the system was overloaded and it is no longer overloaded.

as shown here:

```

void platform_main (void)
{
    reset_reason = ACTIVE_LOW(mon_reset_reason()) & REASON_ALL; /* 1 */
    if (reset_reason == REASON_ALL)
        reset_reason = 0;

    r4k_get_revision_string(proc_revision);                      /* 2 */

    kernel_set_sched_loop_hook(c7100_sched_loop_end);           /* 3 */
}

```

7.2.3.2 AS5800 “Nitro” platform_main()

The following example shows how the Cisco AS5800 “Nitro” Access Server Router uses [platform_main\(\)](#) to perform fundamental initialization. The Cisco AS5800 [platform_main\(\)](#) routine is found in the `src-4k-nitro/platform_nitro.c` source file.

This hook allows the AS5800 to:

- 1 Verify that this image (the CICL Chassis Interface and Common Logic board) matches the hardware. If it doesn’t match, the called routine will `crashdump()`.
- 2 Disable all interrupts by setting the system FPGA register. This routine will also flush the I/O cache.
- 3 Retrieve the reason for the last reset.
- 4 Build the CPU revision string in `proc_revision()`.
- 5 Set the end-of-scheduler routine, `nitro_sched_loop_end()`. This routine will periodically trigger network interfaces “reawakening” if they were previously prohibited from receiving packets (“throttled”) because the system was overloaded and it is no longer overloaded.

as shown here:

```
platform_main (void)
{
    (das_platform->check_das_card)(); /* 1) */

    nitro_disable_all_interrupts(); /* 2) */

    reset_reason = ACTIVE_LOW(mon_reset_reason()); /* 3) */

    r4k_get_revision_string(proc_revision); /* 4) */

    kernel_set_sched_loop_hook(nitro_sched_loop_end); /* 5) */
}
```

7.3 main() -> Memory Initialization

One of the most critical parts of a platform's initialization is memory initialization within the network device. The function hook `platform_memory_init()` allows platforms to declare memory regions, including those dynamically allocated, and pools to the system code using the functions supplied by the memory management code. These functions are described in the "Memory Management" chapter.

7.3.1 platform_memory_init() Entry Conditions

As discussed in the previous section, `platform_main()` is the first platform-specific function called by `main()`, performing only the most rudimentary initialization actions and dealing primarily with hardware reset and verification. After this routine is finished, a checksum of the image is generated and the List Manager is initialized.

Next, the routine `memory_init()` is called. This function will use the List Manager to initialize the head of the list for regions and it will call the next platform-specific function, `platform_memory_init()`.

The `platform_memory_init()` function depends on results from the `platform_main()`.

7.3.2 platform_memory_init() Tasks

The main task of `platform_memory_init()` is to establish the various regions and also memory pools to be used for `malloc()`, as follows:

- Create all memory regions.
- Create memory pools for all regions used for dynamic memory allocations.
- Assure that all mandatory memory pools are created or aliased.
- Set up alternate or "fallback" memory pools as appropriate.

In addition, `platform_memory_init()` may perform additional tasks such as:

- Retrieve from the hardware the amount of memory on this particular system, via `mon_getmemsize()` and validate that the amount is above the minimum requirements.
- Retrieve the amount of IO memory via `mon_get_iosize()`.
- Carve a single bank of memory into several regions; for example, 50% to processor memory and 50% to IO memory.

- Reset the hardware memory access structures, the TLBs (Translation Lookaside Buffers).
- Map IO memory into cached and uncached addresses, for example with the MIPS processor series.

For example, the [platform_memory_init\(\)](#) platform-specific function can be as follows:

```
... main() continued ...
platform_memory_init()      platform_xxxx.c
    /* for example:
     my_memory = mon_get_memsize()
     my_iomem = my_memory_calculation()
     region_create()      divides image text, data, BSS, and heap into three regions - dynamically
                           allocated, pmem, and iomem
     mempool_create(pmem;iomem)  platforms have two banks of processor-only memory, or one
                                bank of processor-only memory and one bank of iomem, or
                                one bank divided into processor-only and iomem (for example,
                                the 2600 and the 3600)
     mempool_add_alias(FAST;PSTACK;ISTACK;PCIMEM)  aliases to memory pools; FAST is
                           aliased to pmem, PSTACK and ISTACK are aliased to
                           processor-only memory, and PCIMEM (PCI bus memory) is
                           aliased to local pmem
     chunk_init()
     .
     .
     .
     ... main() continues ...
```

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.

If the platform does not support either `coredump` or `iomem`, then the corresponding `coredump_supported` and `iomem_supported` flags need to be reset to FALSE in [platform_memory_init\(\)](#). These flags are set to TRUE by default.

The routine `memory_init()` finishes by calling `chunk_init()` to allow the chunk manager to initialize. Returning to `main()`, this routine will call `list_memory_init()` to establish “List Elements” which are used to string together items to build lists and queues. At this point, the following functions are available:

- `malloc()`
- `chunk_create()` and `chunk_malloc()`
- `list_create()` and `list_enqueue()`

7.3.3 Example: Memory Initialization

The following example shows how to implement [platform_memory_init\(\)](#).

7.3.3.1 ubr900 platform_memory_init()

The following example of using [platform_memory_init\(\)](#) shows the memory initialization code for the Cisco 950 SOHO Small Office/Home Office Router platform. The Cisco 950 [platform_memory_init\(\)](#) routine is found in the `src-m850-c900/platform_c950.c` source file.

This hook allows the Cisco 950 to:

- Retrieve the system memory size and define a region for all memory.

- Add regions for the image's text, data and BSS segments.
- Carve the remaining memory between the processor and IO, and create regions and memory pools for the processor and I/O dynamic memory areas.
- Create aliases for other mandatory memory pools.

as shown here:

- 1 The first section of the memory initialization code interrogates the ROM monitor to find out how much memory is installed. It also divides the available memory into IO memory and processor memory, as follows:

```
void platform_memory_init (void)
{
    ulong memsize, io_memsize;
    memsize = mon_getmemsize();
    /*
     * As a default, 75% of the DRAM will go to text, data, and heap,
     * while 25% will go to I/O memory.
     */
    io_memsize = mmu_get_iomem_size();
    memsize -= io_memsize;
```

- 2 The next section of the memory initialization code 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 most Cisco products.

It defines additional 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.

```
region_create(&main_region, "main", (void *)RAMBASE, memsize,
             REGION_CLASS_LOCAL,
             REGION_FLAGS_DEFAULT | REGION_FLAGS_EXCEPTION_DUMP);

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);
```

- 3 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. For this platform, the mandatory memory pool "fast" does not exist, so it is aliased to "heap".

```
region_create(&pmem_region, "heap",
             _END, ((ulong)main_region.end - (ulong)_END),
             REGION_CLASS_LOCAL, REGION_FLAGS_DEFAULT);
mempool_create(&pmem_mempool, "Processor", &pmem_region, 0,
               NULL, 0, MEMPOOL_CLASS_LOCAL);

mempool_add_alias_pool(MEMPOOL_CLASS_FAST, &pmem_mempool);
```

- 4 The IO memory region, used for packet data buffers and other interface-related structures, is at the end of the one memory bank for this small network router. The following code declares the last portion of memory to IO memory.

In addition, this series requires a memory pool of type PCIMEM. Because there is none on this platform, it is aliased to IOMEM. “PCIMEM” refers to memory which is accessible to the network interfaces via a PCI bus. The reason for the two designations is a holdover from the switchover period from direct-accessed memory (which required consideration of endianness) and PCI-accessed devices.

```
region_create(&io_region, "iomem",
             (void *)c950_mmu_tbl[C950_MMU_TBL_IOMEM_ENTRY].vbaseaddr,
             ADRSPC_IO2_SIZE,
             REGION_CLASS_IOMEM, REGION_FLAGS_DEFAULT |
             REGION_FLAGS_EXCEPTION_DUMP);

mempool_create(&iomem_mempool, "I/O", &io_region, C950_MEMPOOL_ALIGN,
               freelist_iomem_block_sizes, nfreelist_iomem_block_sizes,
               MEMPOOL_CLASS_IOMEM);

mempool_add_alias_pool(MEMPOOL_CLASS_PCIMEM, &iomem_mempool);
```

- 5 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 two memory pools are a carryover from times long past, but the requirement for their definition has never been removed.

```
mempool_add_alias_pool(MEMPOOL_CLASS_ISTACK, &pmem_mempool);
mempool_add_alias_pool(MEMPOOL_CLASS_PSTACK, &pmem_mempool);
}
```

This completes [platform_memory_init\(\)](#) for the Cisco 950.

7.4 main() -> Exception Initialization

Next, the platform hook [platform_exception_init\(\)](#) is provided to initialize the platform’s exception and interrupt handlers (up to seven levels of interrupts are allowed). For port adapters (pas), set two interrupt levels: one for standard packets and one for network management issues. 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 platform_exception_init() Entry Conditions

The [platform_exception_init\(\)](#) function depends on results from the [platform_memory_init\(\)](#) function.

7.4.2 platform_exception_init() Tasks

Upon completion of [platform_exception_init\(\)](#), all interrupt handlers above LEVEL_CONSOLE must be established. Because the exact order of LEVEL_CONSOLE varies, one or more interrupt routines may have to be established. The required tasks are:

- Disable interrupts.
- Create a new interrupt level (LEVEL_ERROR), specifying the routine to handle error interrupts.
- Create additional interrupt levels, such as:
 - LEVEL_NMI for the CPU-specific NMI (Non-Maskable Interrupt) handler.
 - LEVEL_MBUS for the Maintenance Bus used for distributed communications.
 - LEVEL_EHSA for EHSA (Enhanced High System Availability) communications between master and standby RP.
 - QUICC_IPL_LEVEL7 for Motorola PPC Power PC PIT (Programmable Interval Timer).

Examples of some additional, typical tasks are:

- Clear any previously-posted exceptions and interrupts, including OIR, Network Management, Network I/O.
- Initialize the background clock exception handler.
- Reset various hardware components.

Examples from various platforms are: the CPU cache, system controller, PCMCIA controller, any PCI buses, IO FPGA (Input/Output Field-Programmable Gate Array), Switch Fabric ASICs, and CSAR (CiscoCell Segmentation & Reassembly) chip.

- Obtain and save information from various hardware components.

Examples of information obtained on various platforms are: system cookie, reset reason, default MAC addresses, chassis serial number, and PA EEPROM.

Examples of some less common tasks are:

- Provide configurations to hardware components.

Examples from various platforms are: the system controller, PCI bus characteristics, PIT (Programmable Interval Timer) refresh interval.

- Validate software and hardware configuration.

For example, verify the CPU level and the system clock speed, detect the required interfaces and the “toaster” PXF (Parallel eXpress Forwarding) engine, and use the interface types to determine the particle size.

- Establish software structures for hardware components.

Examples include Switch Fabric control ASICs and the slot conversion table.

- Select a particle size based on the network interfaces on this particular system.
- A few platforms set the system status LED here instead of in platform_main().

For example, the `platform_exception_init()` platform-specific function can be as follows:

```
... main() continued ...
platform_exception_init()          platform_xxxx.c
    /* for example
     * createlevel(ERROR)
     * createlevel(MY_PLATFORM_SPECIFIC)
     * init_clocktick_handler() /* NMI needed for non maskable interrupt handler (NMI) timer
     * setup
     * my_hardware_init()
         my_system_controller_init() initializes the controller
         my_pci_bus_init()      initializes the pci bus
         my_pcmcia_init()      initializes the pci card
```

```

my_verify_hardware_matches()    checks that the software matches the hardware
my_choose_particle_size()      selects the particle size based on the interface
my_ehsa_interrupts_disable()   used for HA to allow another operating system to take over
... main() continues ...

```

7.4.3 Examples: Exception Initialization

The following examples show how to implement [platform_exception_init\(\)](#).

7.4.3.1 c1000 platform_exception_init()

The following example of using [platform_exception_init\(\)](#) shows the Cisco 1x00 series (c1400 & c1600) exception initialization. The Cisco 1x00 series [platform_exception_init\(\)](#) routine is found in the `src-36-1es/platform_c1000_comn.c` source file.

This hook allows the Cisco 1x00 to:

- 1 Initialize the background clock exception handler via a CPU-specific routine.
For the Motorola CPU, this is pretty easy—update the CPU exception table with the clock exception handler routine.
- 2 Perform CPU-specific interrupt level setup via another CPU-specific routine.
- 3 Establish the PIT (Programmable Interval Timer) to the time specified via `platform_get_value(PLATFORM_VALUE_REFRESH_TIME)`.
- 4 Turn on the “system ok” LED.

as shown here:

```

void platform_exception_init (void)
{
    init_clocktick_handler();           /* 1) */
    stack.hardware_init();             /* 2) */
    c1000_init_pit_timer();           /* 3) */
    c1000_post_exception_init();      /* 4) */
}

```

7.4.3.2 c7140 platform_exception_init()

The following example of using [platform_exception_init\(\)](#) shows the Cisco 7140 EGR Enterprise Gateway Router exception initialization. The Cisco 7140 [platform_exception_init\(\)](#) routine is found in the `src-4k-egr/platform_egr.c` source file.

This hook allows the Cisco 7140 EGR to:

- Disable all interrupts.
- Establish the clock exception handler.
- Reset various hardware components.
- Verify the hardware configuration & components.

as shown here:

- 1 The first section of the exception initialization code is similar on almost all platforms, and this is to initialize the background clock exception handler. The routine `init_clocktick_handler()` is defined for each CPU type, such as the MIPS CPU used with the Cisco 7140. The MIPS routine will:

- Disable all interrupts, just to be safe.
- Set up a new interrupt level for NMI (Non-Maskable Interrupt).
- Update the CPU Exception Table with the CPU interrupt handler routine.
- Call the CPU-specific routine to clear both the Instruction and Data CPU caches.
- Set the clock to 4ms ‘tick’ and set the watchdog counter for 16ms.
- Reenable watchdog interrupt.

```
void platform_exception_init (void)
{
    init_clocktick_handler();
```

- 2 The next section of the exception initialization code, applicable to all c7200-based platforms, including the c7140 EGR, will obtain the CPU type by reading the EEPROM and obtaining the correct “c7200 configuration table”. The c7140 EGR has only one hardware layout, so only one entry in the table. In contrast, the C7200, an older, evolving platform, has almost 10 entries.

If a corresponding table is not found, the routine will post an error message and reload the ROM Monitor image. Notice that the normal console display is not used as the console is not yet initialized.

```
/*
 * Get the cpu cardtype and based on the cardtype
 * find the system configuration structure.
 */
cpucard_type = c7100_cpucard_type();

c7100_config = platform_get_system_config(cpucard_type);
if (c7100_config == NULL) {
    /*
     * Unknown NPE. Crash and burn.
     */
    console_message ("\n%%Error: Unrecognized EGR card\n");
    mon_reload();
}
```

- 3 Next, the c7140 exception initialization routine will retrieve the MP midplane `ID_PROM`. Cisco devices have this chip which describes characteristics such as the number of slots and the default base MAC address as well as serial numbers (chassis and motherboard) and CPU and power supply type designation.

The data retrieved from the `ID_PROM` is copied into the generic field `hw_serial` to be retrieved later by `platform_get_string(PLATFORM_STRING_HARDWARE_SERIAL)`.

```
/*
 * Get the MP eeprom info.
 */
c7100_get_mp_type();

/*
 * Format the serial number.
```

```
 */
sprintf(hw_serial, "%u", c7100_chassis_serial);
```

- 4 The next step taken by the c7140 exception initialization is to verify the configuration, running a function vector from the CPU-specific table obtained earlier. For the c7140, that routine is `platform_egr_config()`. This routine has only one configuration characteristic to set, the maximum number of words in a single burst.

Again, if this routine were to return an error, the routine will post an error message and reload the ROM monitor image using the special pre-console-initialization display routine.

```
/*
 * Verify and adjust the system configuration table
 * based on the MP.
 */
if (! (c7100_config->verify_config)(c7100_config)) {
    /*
     * Unsupported MP. Crash and burn.
     */
    console_message("\n%%Error: Unrecognized midplane in chassis\n");
    mon_reload();
}
```

- 5 The last step by the c7140 exception initialization is to call the general C7200 exception initialization routine to:
- Create a new interrupt level (`LEVEL_ERROR`), specifying the routine to handle error interrupts.
 - Reset the PCMCIA card.
 - Reset any pending OIR, Network I/O, and Midplane Management interrupts.
 - Initialize the PCI subsystem, including the system controller, the bridges to the PCI buses, and the PCMCIA's PCI controller.

```
/*
 * Initialize Predator exception handlers
 */
c7100_exception_init();
```

This completes `platform_exception_init()` for the Cisco 7140 EGR Enterprise Gateway Router.

7.5 main() -> Console Initialization

The last part of a platform's initialization before the `init_process()` initialization is the platform's console initialization, `console_init()`.

7.5.1 console_init() Entry Conditions

The `console_init()` function depends on results from the `platform_exception_init()` function.

7.5.2 console_init() Tasks

The `console_init()` function performs the following tasks (most of the console initialization code is similar on almost all platforms):

- Initialize the console hardware or UART (Universal Asynchronous Receiver/Transceiver).
- Fill in the static “console vector” structure with platform-specific function vectors.
- Allocate and initialize the TTY structure.
- Create a new `LEVEL_CONSOLE` interrupt level and specify the console interrupt handler.

The `console_init()` function allows the platform’s console to be reset and to be set up, as follows:

```
... main() continued ...
    console_init()      cxxxx_console.c
        createlevel(CONSOLE,c3600_cons_int)   creates the interrupt level for the console
        my_console_vectors_init()
        tty_init()
        serial_setbuffers()
        my_setup_console_hardware()
        scheduler_init()
            process_set_boolean(watched_system_init_boolean,FALSE)
        subsys_init()
        registry_init()
        print_restricted_rights()
            cisco Systems, Inc.
            170 West Tasman Drive
            San Jose, California 95134-1706

Cisco Internetwork Operating System Software
IOS (tm) 3600 Software (C3640-I-M), Version 12.2(0.14), BETA TEST SOFTWARE
Copyright (c) 1986-2001 by cisco Systems, Inc.
Compiled Mon 05-Mar-01 16:50 by pwade
Image text-base: 0x600089A8, data-base: 0x60A48000    specifies the address of text-base
and data-base
    scheduler()
... main() continues ...
```

After the console has been set up, the scheduler is initialized with the `scheduler_init()` function. Then, the `subsys_init()` function goes through the data section in order to determine the class and the subsystem initialization sequence, the `registry_init()` function sets up the internal structures to allow `reg_add` to complete, the `print_restricted_rights()` function prints the restricted rights, and the `scheduler()` function takes over.

When the `main()` function transfers control of the system to the scheduler, the scheduler does not return until the system is shut down. The scheduler’s responsibility is to select processes to run and pass control of the CPU to the selected process. The scheduler selects `init_process()` to run.

7.5.3 Example: Console Initialization

The following example shows how to implement `console_init()`.

7.5.3.1 c2600 console_init()

The following example using `console_init()` is a more complex implementation that shows the Cisco 2600 Multiservice Router “quake” console initialization. The Cisco 2600 `console_init()` routine is found in the `src-m860-c2600/c2600_console.c` source file.

Most of the console initialization code is similar on almost all platforms. This is a typical implementation.

- 1 The first step is quite basic; disable all interrupts from the console as well as any TTY or VTY terminals.

```
tt_soc *console_init (void)
{
    register tt_soc *tty;
    vectortype *vector;
    leveltype prev_level;
    prev_level = raise_interrupt_level(TTY_DISABLE);
```

- 2 The next step is to fill in the console vector structure with the exact routines to be called for various conditions. The `vector_init()` function initializes all vectors and upon return, this routine overrides them.

```
/*
 * Initialize the vector pointers
 */
vector = &console_vector;
vector_init(vector);

/* TTY Driver specific calls */
vector->vgetc = serial_getc;
vector->vclearinput = serial_clearinput;
vector->vinputpending = serial_inputpending;
vector->vputc = serial_putc;
vector->vclearoutput = serial_clearoutput;
vector->voutputpending = serial_outputpending;
vector->voutputblock = serial_outputblock;
vector->voutputxoffed = serial_outputxoffed;

/* Modem related function calls setting */
vector->vautosetspeed = async_autosetspeed;
vector->vauto_baud = async_autobaud;
/* Generic function call settings */
vector->vputpkt = generic_putpkt;

/* Helper service entry points */
vector->vservicehook = helper_servicehook;
vector->vnoservicehook = helper_noservicehook;

/* NSC 16552 driver for console needs to provide these calls */
vector->vstartoutput = nsc16552_startoutput;
vector->vstartinput = nsc16552_startinput;
vector->vsetspeed = nsc16552_setspeed;
vector->vstopbits = nsc16552_stopbits;
vector->vparitybits = nsc16552_parity;
vector->v.databits = nsc16552_databits;
vector->vdtrstate = nsc16552_aux_dtr_state;
vector->vsetflow = nsc16552_setflow;
vector->vstopoutput = nsc16552_tx_disable;
vector->vmodemsignal = nsc16552_modemsignal;
```

```

vector->vmodemchange = nsc16552_modemchange;
vector->vportdisable = nsc16552_portdisable;
vector->vportenable = nsc16552_portenable;
vector->vsendbreak = nsc16552_sendbreak;
vector->vfile_transfer_setup = nsc16552_xmodem_setup;

```

- 3 Then, the generalized routine is called to allocate and initialize the TTY. If memory allocation fails in this routine, a crashdump is taken. The input and output buffers are allocated.

```

/*
 * Set up the TTY data structure
 */

tty = (tt_soc *) tty_init(0,READY,CTY_LINE,vector);
if (!tty) {
    reset_interrupt_level(prev_level);
    return (NULL);
}

/*
 * Set num input and output buffers for this driver and for this port.
 */
serial_setbuffers(tty, MAXINCHARS, CONSOLE_OUTCHARS);

```

- 4 Establish the LEVEL_CONSOLE interrupt level and install the appropriate interrupt handler for the platform's console. Notice the interrupt handler nsc16552_console_interrupt() is specific to the particular console interface type.

```

/*
 * Install the 16522 interrupt handler.
 */
m860_install_handler(PQUICC_VECT_IRQ1, LEVEL_CONSOLE,
                      nsc16552_console_interrupt, ED_LEVEL | WM_OFF,
                      "16552 Con/Aux Interrupt");

```

- 5 Initialize the hardware and also update more of the TTY structure, such as the mmap'd address of the Rx buffer and the status register.

```

/* Initialize the UART for the console port. */
Init_UART(tty);

tty->txfinish = nscl6552_tx_disable;
tty->modem_type = MODEM_TYPE_NONE; /* to be consistent with 2681 */

```

- 6 Restore the interrupt level and return the associated TTY structure to main().

```

/*
 * Reset the interrupt level back to the original
 */
reset_interrupt_level(prev_level);

if (mon_get_passwd_protect_mode() == PASSWD_PROTECTION_ENABLE) {
    console_ignore_break();
}

return (tty);
}

```

7.6 init_process() -> Packet Buffer Initialization

The first platform-specific function used in the `init_process()` is the [platform_buffer_init\(\)](#) function.

7.6.1 platform_buffer_init() Entry Conditions

Before calling any platform-specific code, the `init_process()` subroutine `system_init()` will perform a number of tasks:

- Disable interrupts of `NETS_DISABLE` and lower.
- Indicate we have not yet started processing the configuration file as yet.
- Set up system clock services.
- Build string for “reason last rebooted”.
- Call `pool_init()` to initialize Pool Manager resources: queue heads, add a function to the `net_onemin` registry service, create a managed queue to address pool expansion requests and create the process to watch that queue.
- Call `pak_pool_init()` to initialize:
 - Establish structures required for “elements”, which are used in building lists and queues.
 - Create headers for various queues used to manage Pool queues, such as Public Pools, Private Pools and Header pools.
 - Add function to the `net_onemin` registry service to later handle particle pool expansion and shrinkage.
 - Create the 6 public pool structures: Small, Middle, Big, VeryBig, Large, and Huge.

Then, the `init_process()` calls the platform-specific function `platform_buffer_init()`. This function depends on the pool manager having been set up, including headers for pool queues and registration to various periodic services, and most importantly, that the six public pools have been established but not populated.

7.6.2 platform_buffer_init() Tasks

The [platform_buffer_init\(\)](#) function performs the following tasks:

- Populate the six public packet buffer pools with the number of packet buffers required by the particular platform based on the operating mode of the device:
 - `PLATFORM_BUFFER_NORMAL`—This is the normal operating mode image for an IOS network device.
 - `PLATFORM_BUFFER_BOOTSTRAP`—This is a bootstrap image, rather than an image for normal network device operation.
 - `PLATFORM_BUFFER_ROUTING`—This is the routing image on switch, for example, the LS2080 ASP (ATM Switch Processor).
- Create a queue of particles to be used later or cloned particles.
- The low-end 800 and 900 series creates additional public buffer pools of unique sizes.

- The Catalyst 6k series creates the global “fast switch” particle pool here. The “fast switch” particle pool is available for acquisition of particles during the interrupt-level forwarding or “fast switching” path. An example is multicasting, where the first particle of a packet is cloned to allow transmission on multiple interfaces due to multicasting.
Most platforms create this in `platform_interface_init()`.
- Increase the number of packet-enqueuing “elements” from the default, basing it on the number of packets created in this routine.
- The ATM Switch platforms, such as the Cat6k “Aries” MSM (Multiyear Switch Module) and C6400 NSP (Network Switch Processor), destroy the public pools just created and create their own of sizes more appropriate for the ATM Switch.

After the `platform_buffer_init(PLATFORM_BUFFER_NORMAL)` routine and the platform-specific buffer allocation has run, a number of other system components will be initialized, such as:

- General IOS system facilities including the console logging facility and the Parser CLI (Command Line Interface)
- Network Applications including AAA Authentication and Authorization & Accounting, ACL Access Control Lists
- Generalized IOS Network Services such as: creating network queues and registering default routines.

After these, all the `SUBSYS_CLASS_KERNEL` subsystems are initialized.

Also, the `set_crashinfo_bufsize()` function is generally called by the platform code during system initialization, after the `platform_buffer_init()` function has been called and before the `nv_init()` function is called.

Thus, the `platform_buffer_init()` function sets up packet buffers with a buffer size from 100 bytes to 18 Kbytes and sets up pools of `paktype` structures. Here is an example of the packet buffer initialization code:

```
... main() continued ...
init_process()
    pool_init()
    pak_pool_create()
    platform_buffer_init()      platform_xxxx.c
        pool_adjust(SM;MID;BIG;VBIG;LARGE;HUGE)
        /* or on core ATM platforms
        pool_destroy(SM;MID;BIG;VBIG;LARGE;HUGE)
        pak_pool_create(SM;MID;BIG;VBIG;LARGE;HUGE)
        pool_adjust(SM;MID;BIG;VBIG;LARGE;HUGE)    Huge is for ATM switch platforms
        /* if using particles
        particle_clones_init()    clone (duplicated first particle) is for multitasking

        - three pools created:
            fs_particle_pool = particle_pool_create()    used for all fastswitch interfaces
            io_particle_pool_create()
            header_pool = pak_pool_create()    allows a new frame for a different MAC address
... main() continues the init_process()...
```

7.6.3 Example: Packet Buffer Initialization

The following example shows how to implement `platform_buffer_init()`.

7.6.3.1 c6400 NRP Network Route Processor platform_buffer_init()

The following example using `platform_buffer_init()` is a standard implementation that shows the Cisco 6400 Distributed Network Device's NRP Routing Processor packet buffer initialization, including packet buffer population and setup of a queue of particles to be used for "cloning". The Cisco 6400 NRP `platform_buffer_init()` routine is found in the `src-m4k-nrp/platform_nrp.c` source file.

Depending on the operating mode passed as the first argument, this routine will allocate a set number of packet buffers for each of the six Public Buffer Pools.

After populating the buffer pools, this `platform_buffer_init()` routine will unconditionally create a queue of particles to be used later for cloned particles.

- 1 Switching on the input argument, if the calling routine specifies that this is only the small bootstrap program, the six packet buffer pools are established with a default of 0 (zero) packets.

```
void platform_buffer_init (platform_buffer_init_type type)
{
    switch (type) {

        case PLATFORM_BUFFER_BOOTSTRAP:
            pool_adjust(small, 0, SMALL_MAX1, SMALL_INIT_BOOT, TRUE);
            pool_adjust(middle, 0, MID_MAX1, MID_INIT_BOOT, TRUE);
            pool_adjust(big, 0, BIG_MAX1, BIG_INIT_BOOT, TRUE);
            pool_adjust(verybig, 0, VERYBIG_MAX1, VERYBIG_INIT_BOOT, TRUE);
            pool_adjust(large, 0, LARGE_MAX1, LARGE_INIT_BOOT, TRUE);
            pool_adjust(huge, 0, HUGE_MAX1, HUGE_INIT_BOOT, TRUE);
            break;
    }
}
```

- 2 If the calling routine specifies that this is the normal production, packet forwarding image, the six packet buffer pools are established with the appropriate default number of packets.

```
case PLATFORM_BUFFER_NORMAL:
    pool_adjust(small, SMALL_MIN, SMALL_MAX1, SMALL_INIT1, TRUE);
    pool_adjust(middle, MID_MIN, MID_MAX1, MID_INIT1, TRUE);
    pool_adjust(big, BIG_MIN, BIG_MAX1, BIG_INIT1, TRUE);
    pool_adjust(verybig, VERYBIG_MIN, VERYBIG_MAX1, VERYBIG_INIT1, TRUE);
    pool_adjust(large, LARGE_MIN, LARGE_MAX1, LARGE_INIT1, TRUE);
    pool_adjust(huge, HUGE_MIN, HUGE_MAX1, HUGE_INIT1, TRUE);
    break;
```

- 3 If this image contains the subsystem `iprouting_init`, this routine is called later in initialization to override the previous settings, providing for a larger number of packet buffers for each public pool.

Notice the code below allows for two different sets of numbers depending on the memory size on the system.

```
case PLATFORM_BUFFER_ROUTING:
    if (mon_getmemsize() == 128 * ONE_MEG) {
        pool_adjust(small, NRP128MB_SMALL_MIN, NRP128MB_SMALL_MAX2,
                    NRP128MB_SMALL_INIT2, TRUE);
        pool_adjust(middle, NRP128MB_MID_MIN, NRP128MB_MID_MAX2,
                    NRP128MB_MID_INIT2, TRUE);
        pool_adjust(big, NRP128MB_BIG_MIN, NRP128MB_BIG_MAX2,
                    NRP128MB_BIG_INIT2, TRUE);
        pool_adjust(verybig, NRP128MB_VERYBIG_MIN, NRP128MB_VERYBIG_MAX2,
                    NRP128MB_VERYBIG_INIT2, TRUE);
        pool_adjust(large, NRP128MB_LARGE_MIN, NRP128MB_LARGE_MAX2,
                    NRP128MB_LARGE_INIT2, TRUE);
```

```

        pool_adjust(huge, NRP128MB_HUGE_MIN, NRP128MB_HUGE_MAX2,
                    NRP128MB_HUGE_INIT2, TRUE);
    } else {
        pool_adjust(small, NRP64MB_SMALL_MIN, NRP64MB_SMALL_MAX2,
                    NRP64MB_SMALL_INIT2, TRUE);
    } else {
        pool_adjust(small, NRP64MB_SMALL_MIN, NRP64MB_SMALL_MAX2,
                    NRP64MB_SMALL_INIT2, TRUE);
        pool_adjust(middle, NRP64MB_MID_MIN, NRP64MB_MID_MAX2,
                    NRP64MB_MID_INIT2, TRUE);
        pool_adjust(big, NRP64MB_BIG_MIN, NRP64MB_BIG_MAX2,
                    NRP64MB_BIG_INIT2, TRUE);
        pool_adjust(verybig, NRP64MB_VERYBIG_MIN, NRP64MB_VERYBIG_MAX2,
                    NRP64MB_VERYBIG_INIT2, TRUE);
        pool_adjust(large, NRP64MB_LARGE_MIN, NRP64MB_LARGE_MAX2,
                    NRP64MB_LARGE_INIT2, TRUE);
        pool_adjust(huge, NRP64MB_HUGE_MIN, NRP64MB_HUGE_MAX2,
                    NRP64MB_HUGE_INIT2, TRUE);
    }
    break;
}

```

- 4 Finally, this routine will unconditionally create a queue of particles to be used later for cloned particles.

```

if (!particle_clones_init(NRP_PARTICLE_CLONES)) {
    crashdump(0);
}

/* end platform_buffer_init() */

```

7.7 init_process() -> Non-volatile Storage Initialization

Next, the platform-specific [nv_init\(\)](#) function sets up non volatile storage and the [subsys_init_class\(\)](#) function brings up KERNEL class subsystems.

7.7.1 nv_init() Entry Conditions

The [nv_init\(\)](#) function depends on results from the [platform_buffer_init\(\)](#) function and expects all SUBSYS_CLASS_KERNEL subsystems to be initialized.

7.7.2 nv_init() Tasks

The [nv_init\(\)](#) function performs the following tasks:

- Establishes the size of the NVRAM for later use.
- Adds in the global configuration command “config-register” nodes.
 - The Chopin VoIP card also adds in the “crashdump” commands.
- Optionally, the configuration file may be “buffered” into conventional heap. To support this
 - Allocate memory for later use in copying the configuration file into the heap.
 - Flag the NVRAM is buffered into heap memory.

- The AS5300 Access Server carves out memory at the top of NVRAM to store the modem table.
- Adds the global configuration command “config-register” nodes to the Parser.
- For MARs (Multiservice Access Routers), erases the configuration file in NVRAM and removes password protection if “return to factory default config” was set.
- The c12000 Line Card indicates no NVRAM by zeroing the size field.

When `nv_init()` is called, a number of system management and general network processes are created:

- `net_background()` that later orchestrates process-level work on behalf of network interfaces.
- `critical_background()` that later calculates system-wide usage and also per network interface resource usage.
- `net_onesecond()` that later orchestrates periodic per-second process-level work on behalf of all manner of IOS applications.

The `nv_init()` function sets up non volatile storage and the `subsys_init_class()` function brings up KERNEL class subsystems, as follows:

```
... init_process() continued ...
nv_init()          cxxxx_nv.c
    /* for example
     my_nvram_size =
     my_malloc_config_memory()
     parser_add_commands()  for config_reg
subsys_init_class(KERNEL)  brings up KERNEL class subsystems
... main() continues the init_process()...
```

7.7.3 Example: Non-volatile Storage Initialization

The following example shows how to implement `nv_init()`.

7.7.3.1 c10k nv_init()

The following example using `nv_init()` shows the Cisco 10000 “Omega” ESR (Edge Services Router) initialization of the NVRAM holding the IOS configuration file. The Cisco C10000 `nv_init()` routine is found in the `src-4k-c10k/c10k.nv.c` source file.

This hook allows the Cisco 10000 to:

- 1 Indicate that a heap copy of the configuration file will be created.
- 2 Obtain the size of NVRAM on this particular system.
- 3 Allocate heap memory to hold the configuration file.
- 4 Add in the global configuration “config-register” command.

```
void nv_init (void)
{
    nv_set_buffered(TRUE);                                /* Step 1) */
    nvsize = c10k_get_sys_nvsize();                      /* Step 2) */
    /*
     * allocate a dram buffer
     * to hold NVRAM contents
     */
    if (!c10k_nvdrum_buffer)                            /* Step 3) */
        c10k_nvdrum_buffer = malloc(nvsize);
```

```

/*
 * Install parser stuff
 */
parser_add_commands(PARSE_ADD_CFG_TOP_CMD, &pname(configreg_command),
                     "configreg"); /* Step 4 */
}

```

7.7.4 platform_nvvar_support()

In addition, for NVRAM variable support per platform, you should add the [platform_nvvar_support\(\)](#) function after the [nv_init\(\)](#) function.

For example, the [platform_nvvar_support\(\)](#) function is used in the `sys/src-4k-c3600/c3600_nv.c` file to indicate support for the `BOOT_ROMMON_VAR` variable:

```

boolean platform_nvvar_support (int var)
{
    if (var == ROMMON_BOOT_VAR)
        return (TRUE);
    else
        return (FALSE);
}

```

7.8 init_process() -> Interface Initialization

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 of the driver subsystems for their network interfaces.

The goal of any [platform_interface_init\(\)](#) hook should be that no references are made to the actual drivers. This way, the network drivers are implemented as free-standing subsystems and are initialized via the subsystem initialization call.

7.8.1 platform_interface_init() Entry Conditions

The [platform_interface_init\(\)](#) function depends on results from the [nv_init\(\)](#) function. For example, the Non Volatile Storage characteristics are expected to be set, and `net_background()`, `critical_background()`, and `net_onesecond()` are expected to have been created.

7.8.2 platform_interface_init() Tasks

The platform-specific routine `platform_interface_init()` performs a vast variety of tasks and not all platforms will perform all tasks. Additional, optional tasks include:

- Create and populate global particle pools and the global header pool for later use by network interfaces.
- Set external LEDs as defined by the platform:
 - Set system LED to green, by the c3600 MARs (Multiservice Access Router series).
- Initialize the port adapter jump table which will later contain the PA-specific network interrupt routines.
- Add routines to numerous network registries, for example, add support for the DMA coalescing engine.

- Add routines into other system registries, such as the platform-specific memory initialization routine.
 - Set the system default MAC address.
 - For distributed systems, initialize the RP-Line Card bus.

Upon completion of this routine it is expected that all the interrupt levels used by the network interfaces are created. For example:

- LEVEL_NETIO for network traffic.
 - LEVEL_PA_MANAGEMENT for management interrupts from port adapters.

Also, all the SUBSYS_CLASS_KERNEL subsystems have been initialized. Then, before the next platform-specific function is called, the SUBSYS_CLASS_EHSA, SUBSYS_CLASS_PRE_DRIVER, and SUBSYS_CLASS_DRIVER subsystems are initialized.

For example, the `platform_interface_init()` function initializes the network interfaces as follows:

```
... init_process() continued ...
platform_interface_init()          platform_xxxx.c
    createlevel(NETWORK)
    my_interrupt_levels_init() /*SCC, for example
    fs_particle_pool = particle_pool_create()
    pas_init_fspak_pool()
    pas_init_jumptable()   determines type and jump to respective routine
    reg_add_subsys_init_class();
    reg_add_if_final_init(pas/slot/nim_if_final_init)
    reg_add_system_configured()
    reg_add_print_features() /* additions to "show ver"
    reg_add_print_memory()   /* more "show ver"
/* and many more platform-series-specific

.
.

.
.

main() continues the init_process()...
```

The `platform_interface_init()` function calls `reg_add_subsys_init_class()` which adds the driver initialization routine to the `subsys_init_class()` registry that will be invoked by the `reg_invoke_subsys_init_class()` function in `subsys_init_class(SUBSYS_CLASS_DRIVER)`.

Note This information can be specified in the `platform_interface_init()` function or previously in the `platform_buffer_init()` function if needed by the `platform_interface_init()` function.

7.8.3 Example: Interface Initialization

The following example shows how to implement `platform_interface_init()`.

7.8.3.1 IAD2402 platform_interface_init()

The Cisco IAD2402 with multiple channels of data and voice/fax user-side traffic for transport over a single wide-area network (WAN) uplink uses `platform_interface_init()` to perform initialization in preparation for network interface initialization. The Cisco IAD2402 Integrated Access Device `platform_interface_init()` routine is found in the `src-m860-iad2402/platform_iad2402.c` source file.

This hook allows the IAD402 to:

- 1 Install the serial communications controller Interrupt Handler for each of the 4 interfaces.

```
void platform_interface_init (void)
{
    pquicc_dpr_t *dpr = GET_IMMR;

    /*
     * Init the SCC interrupt handlers
     */
    pquicc_install_cpm_exception(dpr, pquicc_scc1_interrupt,
                                  PQUICC_CPIC_INTERRUPT_SCC1);
    pquicc_install_cpm_exception(dpr, pquicc_scc2_interrupt,
                                  PQUICC_CPIC_INTERRUPT_SCC2);
    pquicc_install_cpm_exception(dpr, pquicc_scc3_interrupt,
                                  PQUICC_CPIC_INTERRUPT_SCC3);
    pquicc_install_cpm_exception(dpr, pquicc_scc4_interrupt,
                                  PQUICC_CPIC_INTERRUPT_SCC4);
```

- 2 Add the platform-specific print routines into two print registries.

```
/*
 * Add registry callbacks
 */
reg_add_print_memory(platform_print_memory, "platform_print_memory");
reg_add_print_features(platform_print_features,
                       "platform_print_features");
```

- 3 Set the maximum size of the dialer MIB. The actual size is set by the "dial-control-mib max-size" command.

```
    platform_dial_history_table_length_max = 1200;
}
```

7.8.4 init_process() -> Port Adapter Initialization

Next, the `reg_add_subsys_init_class()` function is called to add the callback for final interface initialization and the `reg_invoke_subsys_init_class()` callback function brings up all specified subsystems for the interface devices. The implementation varies depending on the hardware configuration, such as, `pas` (Port Adapters), `nms` (Network Modules), `slots` or `nims` (the older Network Interface Modules). Each of these implementations is to some extent platform-specific, but they are not well structured with a consistent format like the platform-independent `platform_xxxx` routines.

For those platforms which have the OIRable slot/PA implementation, the fourth platform-specific routine to run under the `init_process()` process is `pas_slots_init()`. As the name implies, this routine will initialize the PA (Port Adapter) "slots" structures before PA's are initialized.

7.8.5 pas_slots_init() Entry Conditions

The `pas_slots_init()` function depends on results from the `platform_interface_init()` function and the `reg_invoke_subsys_init_class()` function must have brought up all specified subsystems for the interface devices.

7.8.6 pas_slots_init() Tasks

The primary task of the platform-specific routine `pas_slots_init()` is to initialize the global `slots[]` table which contains information for each physical slot on the chassis.

Additional tasks are:

- For the Cisco 7200 with the PXF Parallel eXpress Forwarding engine, also initialize the “toaster” hardware at this time.
- The Cisco 6400 NRP Network Switch Processor and NRP2 also have a logical slots table to initialize.
- MXG8850 also initializes the particular Port Adapter containing the ATM cell chip.
- A number of port adapters that contain asynchronous network interfaces will also call the terminal initialization routine both during driver initialization & port adapter insertion, via the `plugin_pa_ft->plugin_attach` port adapter function vector, and when a port adapter is removed, via the `plugin_pa_ft->plugin_detach` port adapter function vector to restore the initialized state of the network device.

For example, in this example the platform-specific `pas_slots_init()` function initializes the port adapters, as follows:

```
... init_process() continued ...
subsys_init_class(EHSA)
subsys_init_class(PRE_DRIVER)
subsys_init_class(DRIVER)
    /* start all DRIVER subsystems    to bring up support for Ethernet and serial lines
       reg_invoke_subsys_init_class()
= pas_subsys_init_class() /* or slot or nim or . . .      has two steps
    pas_slots_init()          platform_xxxx.c
    FOR_EACH_SLOT
        slots[n].xxx = init-value    for example, initialize tables with the number of
interfaces per slot
    pas_analyze_interfaces()
    FOR_EACH_SLOT
        pas_control_reset()    platform_xxxx.c    resets and powers on port adapter
        pas_control_power()
    FOR_EACH_SLOT
        reg_invoke_plugin_pa_builder()    specifies the type of hardware found
        = pa_xxxx_create()    my_xxxx_interface.c    associates function vector table
for xxxx device type per slot
        PA.plugin_ft = &my_interface_ft
    FOR_EACH_SLOT
        PA.plugin_ft.plugin_attach    brings up device
        = pa_xxxx_analyze_pa() /* for example my_xxxx_interface.c
        /* for example
            xxxx_pci_init()    resets pci bus
            xxxx_idb_create()    creates interface descriptor block for each interface
... main() continues the pas_analyze_interfaces FOR EACH SLOT in the init_process()...
```

7.8.6.1 IGX8400 pas_slots_init()

The following example shows how the Cisco IGX8400 URM (Universal Router Module), a fully integrated IOS router blade with 2 T1/E1 (VoIP-capable) and 2 Fast Ethernet ports, uses `pas_slots_init()` to perform initialization of all slots on the device. This `pas_slots_init()` routine is found in the `src-5k-vrxml/platform_vrxml.c` source file.

This hook allows the IGX8400 to loop through the slots table, establishing the default setting for various members of each entry.

```
void pas_slots_init (void)
{
    int slot = 0;

    for (slot = 0; slot < MAX_PA_BAYS; slot++) {
        slots[slot].slot_num = slot;
        slots[slot].eeprom = NULL;
        slots[slot].pasmgt = pas_management_list[slot];
        slots[slot].flags |= BOARD_NONEXIST;
        slots[slot].flags &= ~BOARD_ANALYZED;
        queue_init(&slots[slot].plugin_Q, 0 /* no maximum */);

        /* Initialize the nil plugin and create the linkage between
           plugin and slot */
        plugin_pa_nil_init(&nil_plugin[slot], &slots[slot]);
        slots[slot].plugin_ptr = &nil_plugin[slot].base;
    }
}
```

7.8.6.2 PA Function Calls

In general, the PA functions are implemented in a platform-specific fashion with a default behavior that is supplied in case a platform wants it. The supplied default can also be overridden on a per-platform basis. The following subsections list the invocation and implementation information for some of the PA functions.

7.8.6.2.1 PA Functions Called from Interface Initialization

The following PA functions are generic PA code, or generic PA code with platform-specific variations, that are called from the interface initialization code:

Function	Called Function Implementation	Invocation
<code>pas_is_channelized()</code>	PA-generic routine	Interface Initialization
<code>pas_allocate_fspak()</code>	PA-generic, with platform-specific overrides	Interface Initialization
<code>pas_instance_init_common()</code>	PA-generic, with platform-specific overrides	Interface Initialization

7.8.6.2.2 Platform-Specific Functions Called from Platform or PA Code

The following platform-specific functions are called from the interface initialization code:

Function	Called Function Implementation	Invocation
pas_buffer_mempool()	Platform-specific	Interface Initialization
pas_interfaceFallbackPool()	Platform-specific	Interface Initialization
pas_interfaceHeaderPool()	Platform-specific	Interface Initialization

7.8.6.2.3 PA Functions Called from Platform or PA Code

The following PA functions are generic PA code with platform-specific variations that are called from platform or PA code:

Function	Called Function Implementation	Invocation
pas_init_fspak_pool()	PA-generic, with platform-specific overrides	Platform-specific
pas_platform_if_final_init()	PA-generic, with platform-specific overrides	PA-generic, with platform-specific overrides

7.8.7 init_process() - Memory Allocation

One major hardware transition was from memory directly accessed by the network interfaces to an implementation whereby the interfaces sit on a PCI (Peripheral Computer Interface) bus. In the C7200 and platforms evolved from that transition, the term **PCIMEM** is used to differentiate from directly-accessible memory to the PCI-bus-transversed memory.

When C7200-based platforms have PCI-bus-supported network interfaces, the memory is referred to as **PCIMEM**. This memory pool will then be used for all structures which will be accessible to the network interfaces on the port adapter. This includes the Rx and Tx rings, data buffers, and other structures, such as the interface-defined initialization block.

Note The **PCIMEM** memory pool is not a requirement for using network interfaces residing on a PCI bus. Non-c7200-based platforms may still refer to this memory as **IOMEM**.

7.8.8 platform_pcimempool() Entry Conditions

The `platform_pcimempool()` function depends on results from the `platform_interface_init()` function.

7.8.9 platform_pcimempool() Tasks

To provide a generic interface allowing access to different hardware memory layouts, call `platform_pcimempool()` to obtain the memory pool to use for PCIMEM as shown here:

```
uint platform_pcimempool(unit slot);
```

The platform-specific `platform_pcimempool()` function allocates platform-specific memory, as follows:

```
... pas_analyze_interfaces FOR EACH SLOT in the init_process() continued ...
FOR_EACH_SLOT
    reg_invoke_plugin_pa_builder()
= pa_xxxx_create()      my_xxxx_interface.c   associates vector table
                        for the xxxx device type
    PA.plugin_ft = &my_interface_ft
FOR_EACH_SLOT
    PA.plugin_ft.plugin_attach
= pa_xxxx_analyze_pa() /* for example my_xxxx_interface.c
                        /* for example
    xxxx_pci_init()      resets pci bus
    xxxx_idb_create()   create interface description module for each interface
    platform_pcimempool()     platform_xxxx.c   determines pool where
packets will come from
    xxxx_device_reset()   resets device
    xxxx_setup_rx_tx_rings()  creates receive and transmit rings
    pas_activate_bay()    platform_xxxx.c
FOR_EACH_SLOT
    PA.plugin_ft.plugin_attach_completion
    plugin_pa_return_0()  /* frequently this simple
... main() continues the pas_analyze_interfaces in the init_process()..
```

7.8.9.1 Channelized T3 platform_pcimempool()

In the following example, the CT3 (Channelized T3) network interface obtains the system-wide PCI memory pool and allocates the CT3 network interface's Rx ring from that memory pool.

The CT3 port adapter setup is found in the `pas/if_pas_ct3.c` source file.

```
static boolean ct3_create_rx_ring (ct3_vc_t *ds, hwidbtype *idb)
{
    mempool *pool;

    SANITIZE_CT3_INSTANCE(ds, VC_IDB);

    pool = platform_pcimempool(idb->slot);
    if (pool == NULL) {
        return (FALSE);
    }

    /*
     * Allocate rx ring and rx ring shadow
     */
    ds->rxt_malloc = malloc_named_pcimem(pool->class,
                                         sizeof(mxt_dscrptr_t) *
                                         CT3_RX_RING_ENTRIES,
                                         idb->hw_namestring);
    . . . snip . . .
}
```

7.8.9.2 Defining Platform-Specific Memory Pools

The C7200 provides the appropriate memory pool to use for PCI memory allocations:

- If SRAM is defined, use `PCIMEM`.
- For the c7140 EGR (Enterprise Gateway Router) or the c7200 with an NPE300 (Network Performance Engine 300), use `IOMEM`.
- Otherwise, call the platform-specific function `pas_default_buffer_mempool()` to select the memory pool to use.

For example, the Cisco 7200 `platform_pcimempool()` platform-specific function is found in `src-4k-c7100/c7100_memory.c` source file.

```
mempool *platform_pcimempool (uint slot)
{
    /*
     * If we have packet SRAM, that's the one.
     */
    if (c7100_sramsize > 0) {
        return (mempool_find_by_class(MEMPOOL_CLASS_PCIMEM));
    }

    if ((c7100_config->cpucard_type == C7100_CPU_NPE300) ||
        (c7100_config->cpucard_type == EGR_CPU)) {
        return (mempool_find_by_class(MEMPOOL_CLASS_IOMEM));
    }

    return (pas_default_buffer_mempool(slot));
}
```

7.8.9.3 Slot-Specific Selection of Memory Pools

Another example that selects the memory pool to use based on the slot number is the Cat6k “cygnus” Supervisor Module.

- If the request is for Slot 0, this is the EOBC (Ethernet Out of Band Communications) link in this distributed network device. The EOBC interface is not on a PCI bus, so use standard `IOMEM` for it.
- Otherwise, use `PCIMEM`.

The following Cat 6k “Constellation” “cygnus” Supervisor Module `platform_pcimempool()` platform-specific function is found in `const/cygnus/platform_cygnus.c` source file.

```
mempool *platform_pcimempool (uint slot)
{
    /*
     * For EOBC, return pointer to iomem_mempool
     * For IBC,   return pointer to pcimem_mempool
     */
    return ((slot) ? &pcimem_mempool : &iomem_mempool);
}
```

See 7.15.1 Slotunit, Subunit, and Unit Values for more information on slots.

7.9 init_process() -> Filesystem Initialization

The next function to run under `init_process()` is the `platform_file_init()` function.

7.9.1 platform_file_init() Entry Conditions

The `platform_file_init()` function depends on results from the `platform_interface_init()` function.

7.9.2 platform_file_init() Tasks

The primary task of the platform-specific routine `platform_file_init()` is to establish the filesystem(s) on the flash card and define the default filesystem. The `platform_file_init()` function also performs the following tasks:

- For the Cisco 7200 with the PXF (Parallel eXpress Forwarding) engine, also initialize the “toaster” hardware at this time.

ALTERNATE: For the c10k and the c6400 NRP (Network Route Processor), this function is NULL because it performs this functionality from the `platform_interface_init()` function, so it is available with `SUBSYS_CLASS_EHSA` initialization.

ALTERNATE: For distributed platforms, the line cards may not have a separate filesystem, but rather get configuration data from the IOS Route Processor. In this case, this function will also be NULL.

For example, the `platform_file_init()` function sets up the filesystems and initializes non volatile storage, as follows:

```
... pas_analyze_interfaces in the init_process() continued ...
    platform_file_init()          platform_xxxx.c
        ifs_set_default_directory()  checks if filesystem is on flash or 01 -
frequently returns NULL, except for Token Ring devices
        reg_invoke_if_final_init()
        reg_add_create_idb()       adds platform-specific routers
    ... main() continues the init_process() ...
```

7.9.3 Example: Filesystem Initialization

The following example shows how to implement `platform_file_init()`.

7.9.3.1 AS5850 “Nitro” platform_file_init()

The following example shows how the CiscoAS5850 “nitro” Access Service uses `platform_file_init()` to perform initialization of the flash file systems.

This hook allows the AS5850 to create file systems and set the default. The AS5850 has file systems “slot0”, “slot1” and “bootflash”, with “slot0” set as the default.

```
void platform_file_init (void)
{
    fsid_t fsid;

    /*
     * Create a filesystem for "slot0:" 
     */
    fsid = fslib_ifs_create("slot0", IFS_FLAGS_MEDIA_REMOVABLE);
    ifs_add_prefix(fsid, "flash");

    /*
     * Create a filesystem for "slot1:" 
     */
}
```

```

fslib_ifs_create("slot1", IFS_FLAGS_MEDIA_REMOVABLE);

/*
 * Create a filesystem for "bootflash:"
 */
fslib_ifs_create("bootflash", IFS_FLAGS_NONE);

/*
 * Set the default file system
 */
ifs_set_default_directory("slot0:");
}

```

7.10 init_process() -> Terminal Line Initialization

The next platform-specific routine to run under `init_process()` is `platform_line_init()`. This routine will initialize counters and structures to be used later as terminal lines are established while reading in the configuration file.

7.10.1 platform_line_init() Entry Conditions

The `platform_line_init()` function depends on results from the `platform_file_init()` function.

At this point in initialization all subsystems have been initialized, from `SUBSYS_CLASS_REGISTRY` through `SUBSYS_CLASS_MANAGEMENT`.

ALTERNATE: A number of Port Adapters which contain asynchronous network interfaces will also call this terminal initialization routine both during Driver initialization & PA insertion, via the PA Function Vector `plugin_pa_ft->plugin_attach` and when a PA is removed, via the PA Function Vector `plugin_pa_ft->plugin_detach` to restore the initialized state of the network device.

7.10.2 platform_line_init() Tasks

The `platform_line_init()` platform-specific routine is rather consistent. Here are some details of the tasks completed by `platform_line_init()`:

- Set up the default routine indicating software flow control is not configured. This STUB service will be overlaid if software flow control is available for any async lines on the platform, such as with the AS5800, with a c7206 RS (Router Shelf) with async interfaces and the NextPort interface.
- Initialize a number of terminal line counters and maximums, such as the number of consoles, number of auxiliary consoles and maximum number of VTY lines. These will be used later when terminal lines are configured, for example by the `my-router(config)# line xxxx` configuration command.
- Callback any routines registered to participate in terminal line initialization. Asynchronous subsystems have previously registered for this service during their initialization. These routines will perform actions, such as:
 - Initialize structures for each async line, for example as listed in the `MODEMS[]` array or the `NextPort nd_md_card_info.medem_instance[]` array.

- Establish and populate packet buffer pools for the async interfaces such as the 16-line HMM (Hex Mica Modem) and the MIMIC Compaq Microcom Integrated Modem Network Module.
- Call the common vty virtual terminal initialization routine to add in the async parser commands and update the MODEMS[] table/set up the CLI parser's interface table.
- Then, print the hardware information on the console, for example:

```
cisco 3640 (R4700) processor . . .
Process board ID 12345678
R4700 CPU at 100Mhz, Implementation 33, Rev 1.0
```

You can 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, although no changes are planned in 12.2.

For example, the `platform_line_init()` function sets up the `tty` and `vty` lines, as follows:

```
... init_process() continued ...
    subsys_init_class(PROTOCOL)    PROTOCOL for networking and routing protocols
    subsys_init_class(LIBRARY)    LIBRARY for HTTP, DHCP, and encryption protocols
    subsys_init_class(MANAGEMENT)  MANAGEMENT for SNMP MIBs and parser
commands
    platform_line_init()          platform_xxxx.c
    Print_Hardware()             prints information about the hardware platform
    cisco 3640 (R4700) processor . . .
    Processor board ID 25128407
    R4700 CPU at 100Mhz, Implementation 33, Rev 1.0
    Bridging software.
    X.25 software, Version 3.0.0.
    4 Ethernet/IEEE 802.3 interface(s)
    4 Serial network interface(s)
    DRAM configuration is 64 bits wide with parity disabled.
    125K bytes of non-volatile configuration memory.
    32768K bytes of processor board System flash (Read/Write)
    platform_get_string()        platform_xxxx.c
    reg_invoke_print_features()  /* platform_print_features()
    reg_invoke_print_controllers() /* platform_print_controllers()
    reg_invoke_print_memory()   /* platform_print_memory()
... main() continues the init_process() ...
```

7.10.3 Example: Terminal Line Initialization

The following example shows how to implement `platform_line_init()`.

7.10.3.1 c1000 platform_line_init()

The following example is a typical implementation of `platform_line_init()` that shows how the Cisco 1x00 series (c1400 & c1600) network devices perform terminal line initialization. The Cisco 1x00 series `platform_line_init()` routine is found in the `src-36-les/platform_c1000_comn.c` source file.

- 1** Initialize the `tty_xon` STUB service to the default. When called, this service assumes that protocol translation is not available, and hence will always return that software flow control logic is not necessary.

```
void platform_line_init (void)
{
    /*
     * Some default function registrations
     */
    reg_add_tty_xon(tty_xon_default, "tty_xon_default");
```

- 2** Next, the routine will initialize a number of counters and maximums used later when the configuration commands are read from the configuration file, and any manual entries performed later.

```
/*
 * First discover the devices and count them.
 */
nconlines = 1;
nauxlines = 0;
nvtylines = defvtylines = DEFAULT_NVTYS;

#if ((PLATFORM_MAXLINES) < (DEFAULT_MAXVTPS + 1 + 1))
#error PLATFORM_MAXLINES too small for DEFAULT_MAXVTPS
/* maxvtylines will be default vty line value as this
   platform doesn't support this many vty lines */
maxvtylines = DEFAULT_NVTYS;
#else
maxvtylines = DEFAULT_MAXVTPS;
#endif

/*
 * Assign base indexes into the MODEMS[] data structure.
 */
VTYBase = freeLineBase;
if (protocolconversion)
    maxvtylines = MAXLINES - VTYBase;
ALLlines = nconlines + nvtylines + auxlines + nauxlines;
```

- 3** Next, the routine will callback any routines registered to participate in terminal line initialization. Asynchronous subsystems have previously registered for this service during their initialization. Example actions have been discussed above.

```
/* Added for Async */
reg_invoke_line_init();
```

- 4** Last, the routine will call the generalized function to initialize the virtual terminals within the MODEMS[] table and add in CLI parser commands for asynchronous lines.

```
vty_init();
}
```

7.11 init_process() -> Configuration Verification Initialization

The last platform-specific routine to run under the `init_process()` process is `platform_verify_config()`. It is used for verifying hardware/software configuration compatibility and for final platform-specific configuration work after all other routines have performed setup.

7.11.1 platform_verify_config() Entry Conditions

The `platform_verify_config()` function depends on results from the `platform_line_init()` function.

7.11.2 platform_verify_config() Tasks

The vast majority of platforms do no work:

```
void platform_verify_config(void)
{
}
```

However, a few platforms do perform verification and other “final configuration” tasks, such as:

- The c7200 series and derivatives check the hardware components for compatibility. For example, the NPE175 & NPE225 CPUs cannot run with a system controller of revision GT64120A_B_0_REV or below.
- The c7140 EGR (Enterprise Gateway Router) removes the previously registered T1/E1 **show support** additional displays.
- The ALC ATM line cards with the Catalyst series network devices forces the NVRAM configuration file to “valid” to avoid initial installation Q&A.
- The Cisco 3631 adds code to adjust the slot/port sequence for a WIC slot 0 card to appear as it would on other platforms.

For example, the `platform_verify_config()` function checks, for the last time, that the hardware and software matches for compatibility, and control is returned to the scheduler to allow other processes to run, as follows:

```
... init_process() continued ...
    platform_verify_config()      platform_xxxx.c
    nv_configure()
    process_set_boolean(watched_system_init_boolean,TRUE)
    ttyprintf()   prints message
    Press RETURN to get started!
    process_kill(THIS_PROCESS)   allows other processes to run
... main() continues...
```

7.11.3 Example: Configuration Verification Initialization

The following example shows how to implement `platform_verify_config()`.

7.11.3.1 c7200 platform_verify_config()

The following example shows the C7200 “CPU-system controller” compatibility verification completed by `platform_verify_config()`. The Cisco C7200 series `platform_verify_config()` routine is found in the `src-4k-c7100/platform_c1700.c` source file.

```
void platform_verify_config (void)
{
    cpucard_type = c7100_cpucard_type();

    /*
     * Display warning if using old GT64120 on npe175/225
     */
    if (cpucard_type == C7100_CPU_NPE225 ||
        cpucard_type == C7100_CPU_NPE175) {
        if (gt64k_rev_level < GT64120A_B_0_REV)
            errmsg(&msgsym(RECALLED_NPE, PLATFORM), gt64k_rev_level);
    }
}
```

7.12 init_process() Final Tasks

After the console is set up, the scheduler is initialized with the `scheduler_init()` function.

Next, the `subsys_init()` function goes through the data section order to determine the class and the subsystem initialization sequence.

The `registry_init()` function sets up the internal structures to allow for future `reg_add` functions to complete.

The function `print_restricted_rights()` prints the restricted rights.

The `main()` function transfers control of the system to the orderly scheduler by calling `scheduler()`, which does not return until the system is shut down.

After the `platform_verify_config()` function checks, for the last time, that the hardware and software matches for compatibility, final system initialization is as follows:

After `platform_verify_config()` is called from `system_init()` during the initialization process, the next steps are to:

- Enable for all interrupt levels.
- Allow EHSA systems to perform primary negotiation.
- Read in and process the configuration file.
- Set the “system initialization” Boolean to TRUE, allowing for other processes waiting on system initialization to complete to run.
- Print the message “Print RETURN to get started”.
- Then, the `init_process()` will kill itself.

7.13 Platform-Specific Strings

User interface commands and other network applications running on IOS 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 7-1 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.

The `platform_get_string()` function returns `NULL` if the requested string is not implemented or is not applicable to a platform.

Note Because a pointer is returned to the string required, the string must not be stored on the local stack.

Table 7-1 Platform Strings

Platform String Flags	Description
<code>PLATFORM_STRING_NOM_DU_JOUR</code>	(Mandatory) Model number or name of the platform. Examples are “c2611”, “IAD2420”, and “CWAN Toaster Line Card”.
<code>PLATFORM_STRING_DEFAULT_HOSTNAME</code>	(Mandatory) Default hostname to be used for a platform if none is specified in the configuration. This string also appears in the user interface CLI prompt. Examples are “Router”, “Switch”, and “CWTLC-Slot4”.
<code>PLATFORM_STRING_PROCESSOR</code>	(Mandatory) Name of the processor used by the platform. Examples are “68360,” “R4K”, “R7000”, and “RM5271-250MHz”.
<code>PLATFORM_STRING_PROCESSOR_REVISION</code>	(Optional) Revision number of the processor used by the platform. Not all processors allow this information to be obtained. For example, on MIPS processors, this can be retrieved via the <code>r4k_get_revision_string()</code> function.
<code>PLATFORM_STRING_HARDWARE_REVISION</code>	(Optional) Hardware version of the platform. This is usually the motherboard revision number. Not all platforms allow this to be obtained. For example, the C7200 series gets this from an EEPROM on the motherboard.
<code>PLATFORM_STRING_HARDWARE_SERIAL</code>	(Optional) Serial number of the platform. This is normally held within the cookie. Not all platforms allow you to obtain this number.
<code>PLATFORM_STRING_VENDOR</code>	(Optional) Vendor derivative of the platform. Examples are “Cisco,” “Cabletron,” and “Bay Networks.”
<code>PLATFORM_STRING_HARDWARE_REWORK</code>	(Optional) Hardware rework version of the actual processor board. This is effectively a subrevision of the motherboard revision number.

Platform-Specific Strings

Table 7-1 Platform Strings (continued)

Platform String Flags	Description
PLATFORM_STRING_LAST_RESET	(Optional) The reason for the last hardware reset event. Examples are “power-on”, “watchdog” and “s/w nmi”.
PLATFORM_STRING_BOOTROM_OR_BOOTFLASH	(Optional) Returns the location of the boot program, either “BOOTFLASH:” or “BOOTROM:” or NULL. This is used in the “show version” command.
PLATFORM_STRING_PCMCIA_CONTROLLER	(Optional) Returns the type of the PCMCIA controller. Examples are “CLPD6729” or “None”. This is used to insert into the Flash MIB structure.
PLATFORM_STRING_HTML_DEFAULT_PATH	(Optional) Returns the default path to be used by HTTP. This was originally implemented for the C6400 which defines the default path as “mir-disk0:/nsp-html”.
PLATFORM_STRING_HTML_HOME_PAGE	(Optional) Returns the home page for this network device. This was originally implemented for the C6400 which defines the default path as “6400.html”.
PLATFORM_STRING_HTML_FIRMWARE_VERSION	(Optional) Is defined but not supported nor used by any platform as of this writing.
PLATFORM_STRING_MIDPLANE_VERSION	(Optional) Returns the home page for this network device. This was originally implemented for the Cisco 10k “omega” which obtains this from the EEPROM.
PLATFORM_STRING_PROMPT_PREFIX	(Optional) Returns a prefix identifier for the IOS prompt to indicate the physical location of this IOS. This was originally implemented for the C6400, for example, “Slot1P:” for a Primary RP in slot 1.
PLATFORM_STRING_PROMPT_SUFFIX	(Optional) Returns textual information to indicate this is a “split processor”. To be applied as a suffix. This was originally implemented for the Cat6k, for example, “net-dev-sp” for a split processor, else there will be a NULL suffix.

7.13.1 Example: Calling Platform-Specific Strings

The following is a display excerpt of the “**show version**” EXEC command for the AS5850 “Nitro”:

```
cisco c5850 (r4k) processor
```

The following is an example of calling the `platform_get_string()` function producing the above output, found within the `PrintHardware()` function in the `ui/exec.c` source file.

```
void PrintHardware (void)
{
    . . . snip . . .
    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);
    . . . snip . . .
}
```

7.13.2 Example: Defining Platform-Specific Strings

The following example shows the `platform_get_string()` function that is implemented on the AS5850.

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`.

The AS5850 Access Server “Nitro” `platform_get_string()` routine is found in the `src-4k-nitro/platform_nitro.c` source file.

```
char *platform_get_string (platform_string_type type)
{
    char *value;

    switch (type) {
        case PLATFORM_STRING_NOM_DU_JOUR:
            value = "c5800";
            break;

        case PLATFORM_STRING_DEFAULT_HOSTNAME:
            if (!nitro_ios_name_ptr || (nitro_slot() != nitro_saved_slot)) {
                nitro_saved_slot = nitro_slot();
                sprintf(nitro_ios_name, "%s%d",
                        NITRO_DEFAULT_HOSTNAME_PREFIX, nitro_saved_slot);
                nitro_ios_name_ptr = nitro_ios_name;
            }
            value = nitro_ios_name_ptr;
            break;

        case PLATFORM_STRING_PROCESSOR:
            value = "R4K";
            break;

        case PLATFORM_STRING_PROCESSOR_REVISION:
            value = proc_revision;
            break;

        case PLATFORM_STRING_VENDOR:
            value = "cisco";
            break;

        case PLATFORM_STRING_LAST_RESET:
            value = interpret_reset_reason(reset_reason);
            break;

        case PLATFORM_STRING_HARDWARE_REVISION:
        case PLATFORM_STRING_HARDWARE_SERIAL:
        default:
            value = NULL;
            break;
    }

    return (value);
}
```

7.14 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 7-2 describes some of 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.

Note Check your current code base for a full list of enum values to be requested for your IOS version.

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 <code>PLATFORM_VALUE_SERVICE_CONFIG</code> 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_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 initially disabled.
PLATFORM_VALUE_LOG_MAX_MESSAGES	(Optional) Default maximum number of logging messages (error or debugging) to be queued waiting for display to the console. This value is returned to the logger during system initialization and used when creating the managed queue of messages to be displayed on the console. If the platform routine returns 0, a default of <code>LOG_DEFAULT_QUEUESIZE</code> (100 as of this writing) will be used. This value may be overridden by the hidden logging queue-limit configuration command.

7.14.1 Example: Calling Platform-Specific Values

The following example shows how to call the `platform_get_value()` function. This example is from system startup, the routine `main()`, found in the `os/sched.c` source file.

```
boolean main (boolean loading)
{
    . . . snip . . .
/*
 * Determine our processor and family type.
 */
cpu_type = platform_get_value(PLATFORM_VALUE_CPU_TYPE);
cpu_family = platform_get_value(PLATFORM_VALUE_FAMILY_TYPE);
    . . . snip . . .
}
```

7.14.2 Example: Defining Platform-Specific Values

The following example is an excerpt showing how the system software for the Cisco 1600 dynamically obtains the CPU and family types for this platform.

The Cisco 1600 router `platform_get_value()` routine is found in the `src-36-c1600/platform_c1600.c` source file.

```
uint platform_get_value (platform_value_type type)
{
    uint value;

    switch (type) {
    . . . snip . . .
        case PLATFORM_VALUE_CPU_TYPE:
            value = mon_proctype();
            break;

        case PLATFORM_VALUE_FAMILY_TYPE:
            if (cpu_type == CPU_VOLCANO)
                value = FAMILY_C1600;
            else if (cpu_type == CPU_C1400)
                value = FAMILY_C1400;
            else
                value = FAMILY_UNKNOWN;
            break;

        default:
            value = 0;
            break;
    }

    return (value);
}
```

7.15 Interface ID Format

This section includes information on identifying and specifying interfaces.

A hardware interface is identified by a set of attributes that uniquely identify the interface's location within the system. These attributes are stored within the interface using two separate mechanisms.

In the first mechanism, the hardware identity is stored within an `idb_identity_t` structure (`hwidb->identity`). The second mechanism stores the hardware identity directly within the HWIDB (`hwidb->type`, `hwidb->unit`, `hwidb->slot`, etc.) for quick reference.

The `hwidb->identity` is allocated when the IDB is created by calling one of the following:

- `idb_create_with_identity()`
- `vidb_malloc_with_identity()`

The identity stored within the IDB must be unique within the system. However, the identity associated with any particular IDB may be deleted and a new one assigned using the appropriate APIs.

Previously, a HWIDB of type FOO with a single index of unit was created as follows:

```
hwidbtype *create_foo (uint unit)
{
    hwidbtype *hwidb;

    hwidb = idb_create();

    if (hwidb) {
        hwidb->type = IDBTYPF_FOO;
        hwidb->unit = unit;

        hwidb_get_next_if_index(hwidb);

        // Initialise rest of hwidb
    }

    return hwidb;
}
```

However, a new method is preferred as follows:

```
hwidbtype *create_foo (uint unit)
{
    hwidbtype *hwidb;
    idb_identity_t identity;

    idb_id_clear(&identity);
    idb_id_set_type(&identity, IDBTYPF_FOO);
    idb_id_set_unit(&identity, unit);

    hwidb = idb_create_with_identity(&identity);

    if (hwidb) {
        // initialise rest of hwidb
    }

    return hwidb;
}
```

Note In the previous example, `idb_create_with_identity()` is populating `hwidb->type` and `hwidb->unit`. Therefore the code itself should not try to write these.

The `/vob/cisco.comp/idb-identity/include/idb_identity.h` file contains the attributes that define the identity of a HWIDB.

In general, for slotted platforms, the shelf, slot, slotunit, type, subtype, and vc attributes are used.

For platforms that use PAs or have evolved from PAs, the shelf, slot, subunit, type, subtype, and vc attributes are used.

7.15.1 Slotunit, Subunit, and Unit Values

These values are platform-specific. Interface identification is inherently platform-dependent, because it relies upon the physical arrangement of the platform, but the values used to identify these arrangements are held within a platform-independent data structure, the IDB. Generally, the values are used as follows:

- Slotunit: Identifies a particular hardware interface within a particular slot, for example one Ethernet port on an 8E PA can be represented as Ethernet 2/3.
- Subunit: Identifies a particular virtual channel that is part of a physical interface, for example a channel group on a channelized T1 interface can be represented as Serial 2/3:10.
- Unit: If the interface is a dynamic interface, only the unit number is used. This value is also used on non slotted platforms where the numbering is a simple index (for example c4500, with Ethernet 1). Platforms that have a slotted numbering scheme normally use the slotunit value instead of unit.

Note The platform can kludge its way around the use of these values. When they are used, the slotunit and subunit values commonly encode the subslots and port numbers as well. The RSP identifies interfaces using a slot/unit convention, but the VIP uses a slot/PA-bay/unit convention by converting the PA-bay/unit into a single unit number. For example, on the c10k, the subunit is split such that the least significant bits are the port number and the remaining bits are the subslot. On the c7500, the slotunit encodes the PA bay (similar to a subslot) and the unit within that PA.

Code (especially platform-independent code) should avoid using the slotunit, subunit, and unit values to navigate around interfaces, but instead should be using internal pointers to represent relationships. For example, the old channelized controller code used to walk through the list of interfaces looking for IDBs with matching slot/unit/slotunit(appl_no) values - this was replaced with pointers to the IDB (and the IDBs owned by controllers have a back pointer to the CDB), which make the code independent from whatever navigation scheme is used by the platform.

7.15.2 Specifying Interface Identifier Format

The exact implementation of interface identifiers is wholly dependent on the particular platform. Interface identification is inherently platform dependant, because it relies upon the physical arrangement of the platform. All values used to identify these arrangements are held within a platform independent data structure (the `hwidb`), but with unique representations.

As of this writing, the identification variables used within the `hwidb` are `shelf`, `slot`, `unit`, `subslot` & `subunit` in a wide variety of layout characteristics.

In this section both the three “standard” identifications as well as some alternative implementations are listed in the next three sections.

Note Be aware that a platform can kludge its way around the use of any of these values. Within generalized code you cannot assume any particular implementation.

7.15.3 One-level Standard or “Unit”

The most simple implementation is used on platforms where the network interface identification is a simple index, such as “interface Serial 1”. This identification is used on smaller platforms such as the c800, c900 and c1x00 series devices.

The “unit” identification method uses a single field, placing the unit number in `hwidb->unit`.

This identification method is established by the platform within the `src-*/platform*` source file by setting the generalized function `platform_find_interface` to be `platform_find_unit_interface`, for example in `src-4k-c5300/platform_c5300.c`:

```
boolean (*platform_find_interface)(parseinfo *, iftype *, int *,
                                 idbtype **,
                                 interface_struct * const, char *)
= parser_find_unit_interface;
```

This format is also used if the interface is a dynamic interface, such as virtual IDBs and asynchronous interfaces.

7.15.4 Two-level Standard or “Slots”

The next implementation to be discussed is the “slotted” identification where the “slotted” layout (I.e., slots or Port Adapters with embedded network interfaces). For these devices, the network interface identification is a dual identifier, such as “interface Ethernet 2/1”. Examples of this format are the MARs c2600 & c3600 and the C7200 series.

The “slotted” identification uses two fields: `hwidb->slot` & `hwidb->unit`. For the example of “interface Ethernet 2/1”, the `hwidb->slot` is set to “2” and the `hwidb->unit` is set to “1”.

This identification method is established by the platform within the `src-*/platform*` source file by setting the generalized function `platform_find_interface` to be `platform_find_slotted_interface`, for example with the IGX 8400 URM Universal Router Module in `src-5k-vrxml/platform_vrxml.c`:

```
boolean (*platform_find_interface)(parseinfo *, iftype *, int *,
                                 idbtype **,
                                 interface_struct * const, char *)
= parser_find_slotted_interface;
```

7.15.5 Three-level Standard with “shelf”

The more complex “shelf” identification has a three-identifier format, for example “interface Ethernet 3/2/1”. The first identifier, `hwidb->shelf` indicates the “shelf” on which the interface resides. A “shelf” can be another “slotted” independent Network Device, such as the Cisco 7206 shelf in an AS5850 “Nitro” Access Server. The next two identifiers are the standard “slotted” implementation `hwidb->slot` is set to “2” and `hwidb->unit` is set to “1”.

The Catalyst 3750 also uses this format.

This is set by the platform within the `src-*/platform*` source file by setting the generalized function `platform_find_interface` to be `platform_find_shelf_interface`. For example the c7206 will check if it is a shelf within an AS5850 “Nitro” Access Server in `src-4k-c7100/platform_c7100.c`:

```
void platform_interface_init (void)
{
    . . . snip . . .
/*
 * If we are a shelf, then adjust the interface and controller
 * name routines accordingly. Also create the router shelf
 * object.
 */
if (shelf_indicator) {
    platform_find_interface      = parser_find_shelf_interface;
    platform_create_interface_name = idb_init_shelf_names;
    platform_find_controller     = parser_find_shelf_controller;
    platform_create_controller_name = cdb_init_shelf_names;
    reg_add_parse_controller_slotport(parser_get_controller_slotport,
                                       "parser_get_controller_slotport");
}
. . . snip . . .
}
```

7.15.6 c7500 RSP Three-level Alternative with “slotunit”

The C7500 has two unique components:

- The master processor, the RSP or Route Switch Processor, which manages the complex. It is usually not involved in packet forwarding.
- One or more “Interface Processors”, which are actually IOS systems which manage network traffic. Older examples were customized for the network interface type, such as the AIP ATM Interface Processor. Today, the C7500 uses the VIP Versatile Interface Processor which will support a variety of network interface types.

7.15.6.1 The C7500 Route Switch Processor

The C7500 RSP Route Switch Processor uses the same three-identifier format, “interface Ethernet 3/2/1”. The first identifier, specifies the “interface processor” or the unique board also running IOS, the second indicates the board within that slot and the third the Network Interface. However, the implementation holds both identifiers in a single field `hwidb->slotunit`, in the following manner:

```
if (hwidb->slotunit >= pa_virtual_slot_boundary) {
    pa_bay = PA_BAY_1;
    pa_unit = hwidb->slotunit - pa_virtual_slot_boundary;
} else {
    pa_bay = PA_BAY_0;
    pa_unit = hwidb->slotunit;
}
```

This identification method is established by the platform by setting the generalized function `platform_find_interface()` to be `platform_find_rsp_interface()`, for example in `src-rsp/if_name_rsp.c`:

```
boolean (*platform_find_interface)(parseinfo *, iftype *, int *,
                                   idbtype **,
```

```
interface_struct * const, char *)
= parser_find_rsp_interface;
```

7.15.6.2 The C7500 VIP

The C7500 VIP Versatile Interface Processor has already been identified as an “independent processor” within the C7500. By the time it gets called, it can use the standard “slotted” implementation of hwidb->slot & hwidb->unit.

This identification method is established by setting the generalized function `platform_find_interface()` to be `platform_find_slotted_interface()`, for this platform in `src-svip/platform_svip.c`:

```
boolean (*platform_find_interface)(parseinfo *, iftype *, int *,
                                   idbtype **,
                                   interface_struct * const, char *)
= parser_find_slotted_interface;
```

7.15.7 Catalyst 5K Three-level Alternative with “slotunit”

The Cat5k RSP (Route Switch Processor) uses an alternate identification with a three-identifier format, still using the format “`interface Ethernet 3/2/1`”. The first identifier, `hwidb->slot` indicates a separate “Interface Processor” such as the AIP ATM Interface Processor or the VIP Versatile Interface Processor.

The second identifier `hwidb->subunit` indicates the board within the AIM or VIP and the third identifier `hwidb->unit` indicates the specific Network Interface.

This identification method is established by the platform by setting the generalized function `platform_find_interface()` to be `platform_find_c5rsp_interface()`, for example in `src-c5rsp/if_name_c5rsp.c`:

```
boolean (*platform_find_interface)(parseinfo *, iftype *, int *,
                                   idbtype **,
                                   interface_struct * const, char *)
= parser_find_c5rsp_interface;
```

7.15.8 ONS155xx Optical Network System Alternative

The ONS (Optical Network System) series of platforms uses a variation on the “slotted” interface designation. For these platforms, the following applies.

Three level “slotted” identification as `hwidb->slot & hwidb->subunit & hwidb->slotunit`

This implementation is used on platforms where the network device is a “slotted” layout, with “subcards” identified as physical units which fit into the slot, and the unique network interfaces or “ports” on the subcards are the “slotunit”. This leads to an implementation of “`interface Ethernet 3/2/1`”.

For slots where there is only one subcard, the naming convention is “`interface waveether 3/2/2`”, where `hwidb->slot=3 & hwidb->subunit=2 & hwidb->slotunit=0`. Note that the subunit identifier represents a network interface.

This is set by the ONS15540 platform within the `src-7k-manhattan/platform_manopt.c` source file by setting the generalized function `platform_find_interface` to be `platform_find_manopt_interface`.

```
boolean (*platform_find_interface)(parseinfo *, iftype *, int *,
```

```

        idbtype **,
        interface_struct * const, char *)
= parser_find_manopt_interface;

```

7.15.9 Other Alternate Implementations

A few alternative implementations are listed below.

7.15.9.1 Other usages of hwidb->subunit

The identifier `hwidb->subunit` is used by the AIM (Advanced Integration Module) for sequential identification.

For platforms that have async serial interfaces and use slotted architecture: `hwidb->slotunit = hwidb->unit = tty->ttynum`

7.15.9.2 Documenting your platform identification implementation

If you wish to contribute information on your platform implementation, please send email to ios-doc@cisco.com.

7.15.10 Use other methods for IDB navigation

Code (especially platform independent code) should avoid using these values to navigate the way around interfaces, but instead should be using internal pointers to represent relationships. For example, the old channelized controller code used to walk through the list of interfaces looking for `hwidbs` with matching slot/unit/`slotunit(appl_no)` values - this was replaced with pointers to the `hwidb` (and the `hwidbs` owned by controllers have a back pointer to the `cdb`), which make the code independent from whatever navigation scheme is used by the platform.

7.16 How to Develop New Platform Support

Here are the basic steps for developing IOS support for a new platform:

- Step 1** Create a new source directory. For example, find the directory for an existing platform that includes most of the same hardware as the new platform, and create a new directory:

```
mkdir /path_to_similar_platform/new_platform_directory
```

For example:

```
mkdir /vob/ios/sys/src-m860-c2600
```

- Step 2** Copy the files from the same or a similar type of platform's source directory because it is good to use the tested code.

For example, find an existing platform that includes most of the same hardware as the new platform and copy the files from the existing platform. Copy `console.c` and `console.o` from the existing platform. If the new platform's system controller is similar to another existing platform, copy the system controller files from the other existing platform. And, if the new platform's memory is the same as the memory in another existing platform, copy the memory files from yet another existing platform.

```
cd /vob/ios/sys
cp src-4k-c3600/*console* src-m860-c2600
```

```
cp src-mpc-c800/*nv* src-m860-c2600
.
```

- Step 3** Edit the copied files for the new platform, or if you determine that the existing platform's files are fine as-is for the new platform, reference the existing platform file and delete the copy.

Note Do not mix platform-dependent and platform-independent coding. Compilation fails if mixed platform-dependent and platform-independent coding is detected. Platform-specific functions and definitions are not allowed in platform-independent files.

7.17 How to Request Platform-Specific Action from Platform-Independent Code

The Cisco IOS platform-independent and platform-specific code is tightly coupled and program flow frequently crosses the boundaries between the two types of code. This section describes how the platform-independent code calls the platform-specific code, depending on the action or actions needed.

The general guidelines used to determine how to request platform-specific action from platform-independent code are as follows:

- 1 Check if the action *must* be implemented on all platforms. If so, go to “For a Platform-Specific Action Implemented on All Platforms”.
- 2 Check if the action should *optionally* be implemented on one or many platforms (one-to-one, one-to-none, one-to-many, or many-to-one mappings). If so, go to “Implementing Optional Actions”.
- 3 Check if the action must be implemented on one platform (one-to-one mapping) or if build modularity is a concern (for example, to avoid making the platform-independent code dependent, from the linker’s perspective, on the platform-specific code). If so, go to “For a Platform-Specific Action Implemented on One Platform”.

Note The rationale on whether to use function vectors or tables of function vectors or any of the particular registry types is not distinct to the way to incorporate platform-specific code into an image. It is pretty much the same issue if you have a subsystem that is only linked into some images.

7.17.1 For a Platform-Specific Action Implemented on All Platforms

This section describes the techniques for implementing actions that are required on each platform.

Use a direct function call. This forces the new platform developer to implement the function. An example of this is the `platform_memory_init()` function, which must be defined for each platform:

```
void memory_init()
{
    ... snip ...
    platform_memory_init();
    ... snip ...
}
```

For example, follow these steps:

- Step 1** Define an `extern` function prototype for the action required in a platform-independent `.h` header file. For example, add the `platform_memory_init()` function in `h/platform.h` as shown here:

```
extern void platform_memory_init(void);
```

- Step 2** In the platform-specific code, declare the implementation of the function in a platform-specific `.c` file. For example, the platform-specific `src-4k-c7100/c7100_memory.c` file contains the platform-specific function:

```
/*
 * platform_memory_init:
 * Define and declare all the DRAM memory regions and mempools
 * for this platform
 */
void platform_memory_init (void)
{
    ulong memsize, mem2size, pmem2size;
    ulong iomemstart, iomemsize, iomem2start, iomem2size;
    ulong pmem2start;
    ulong status;
    uint bits35_32 = 0;
    int cputype;
    ulong kseg_mainmem_size = 512 * ONE_MEG; /* max main mem size
accessible thru kseg0/1 */
    #ifdef IOMEMSIZE_ENABLE
    ulong total_req_iomem;
    #endif
    ... snip ...
}
```

- Step 3** Call the platform-specific function from platform-independent code. For example, the `memory_init()` platform-independent function declared in `os/free.c` calls the `platform_memory_init()` platform-specific routine:

```
void memory_init()
{
    ... snip ...
    platform_memory_init();
    ... snip ...
}
```

7.17.2 Implementing Optional Actions

This section describes the techniques for implementing actions that are optional.

There are two basic types of optional actions:

- Functions with a one-to-one mapping (or one-to-none)—For this optional action, create a function vector.
- Functions with a one-to-many or many-to-one mapping—For these optional actions, create a registry (LIST, LOOP, etc., or CASE, RETVAL, VALUE, etc.).

Note Already, there are 645 `ios/sys/*/*.reg` registry files currently existing in 12.3T, so be sure that your registry is not redundant before creating it.

7.17.2.1 For One-to-One or One-to-None Mapping

If implementation is optional and no more than one routine is defined, define a function vector. For information on this type of implementation, see 7.17.3, “For a Platform-Specific Action Implemented on One Platform”.

7.17.2.2 For One-to-Many or Many-to-One Mapping

This subsection includes information on functions with one-to-many and many-to-one mapping. Registries are used for this type of implementation. For more information on registries, see Chapter 14, “Registries and Services.”

7.17.2.2.1 One-to-Many Mapping

For one platform-specific action that needs to be mapped to many platform-independent calls, create a LIST or LOOP registry. This type of registry allows multiple functions to be registered for the same action, so there can be more than one implementation function at a time.

A very simple example is a routine that needs to run every second. IOS implements this per-second process and per-second registry service to perform this:

```
process net_onesecond (void)
{
    . . . snip . . .
    reg_invoke_onesec(net_periodic_suspend);
    . . . snip . . .
}
```

For example, you would follow these steps:

- Step 1** Define a registry with one-to-many characteristics, typically a LIST service, as in this definition within `sys_reg/sys_timer_registry.reg` system timer registry file:

```
DEFINE onesec
    . . . comment . . .
    ILIST
    void
    -
END
```

The LIST service variant ILIST is described below.

- Step 2** Within the platform-specific code, provide the required functionality, as in the c7500 VIP `src-rsp/rsp_vip.c` source file:

```
void vip_crashinfo_scan (void)
{
    . . . snip . . .
}

void vip_txacc_scan (void)
{
    . . . snip . . .
}
```

- Step 3** From a platform-specific function, add the platform-specific routines into the generic registry service, as in the c7500 VIP `src-rsp/rsp_vip.c` initialization routine:

```
static void vipinfo_init (subsysstype* subsys)
{
    svip_crashinfo_init();
    reg_add_onesec(vip_crashinfo_scan, "vip_crashinfo_scan");
    reg_add_onesec(vip_txacc_scan, "vip_txacc_scan");
}
```

- Step 4** In the platform-independent function, call the platform-specific routines via the registry service. For example, platform-independent `net_onesecond()` calls two routines when the `onesec` registry service is invoked for the c7500 VIP:

```
process net_onesecond (void)
{
    while (TRUE) {
        process_sleep_periodic(ONESEC);
        reg_invoke_onesec(net_periodic_suspend);
    }
}
```

The `ILIST` intermediate list service will repeatedly invoke a single registered routine, then invoke the routine passed as the argument to the registry service. This is used for a few IOS generic registry services which would otherwise result in a `CPUHOG` designation.

7.17.2.2.2 Many-to-One Mapping

For many platform-specific actions that need to be mapped to one platform-independent call, create a `CASE`, `RETVAL`, or `VALUE` registry.

An example is the `show diag` enabled command on slotted or port adapter platforms. The routine to be run is selected based on the port adapter type. The c2800 “fellowship” platform assures that the appropriate routine be called based on the particular Voice WIC card installed. So, for multiple actions on a per-platform basis, you can register based on the interface type:

```
static void show_pa_detail(uint pa_bay, boolean override)
{
    . . . snip . . .
    reg_invoke_display_wic_info(slotptr->pa_type, pa_bay);
    . . . snip . . .
}
```

For example, you would follow these steps:

- Step 1** Define a `CASE` registry service, as in this definition within `sys_reg/sys_hw_registry.reg` system hardware registry file:

```
DEFINE      display_wic_info
/*
 * Display the Wan Interface Card info for the specified slot.
 */
CASE
void
int slot
0
int type
END
```

- Step 2** Within the platform-specific code, provide the required functionality, as in the c2800 sys/nms/if_c2800_mbrd_vwic.c voice WIC card source file:

```
void fellowship_mbrd_display_wic_info (int slot)
{
    . . . snip . . .
    show_wic_eeprom_contents(slot, ix);
    . . . snip . . .
}
```

- Step 3** From a platform-specific function, add the platform-specific routines into the platform-independent registry service, as in the c2800 nms/if_c2800_mbrd.c initialization routine:

```
static void fellowship_mbrd_subsys_init (subsysstype *subsys)
{
    . . . snip . . .
    reg_add_display_wic_info(PM_HARDWARE_BOMBOR,
                            fellowship_mbrd_display_wic_info,
                            "fellowship_mbrd_display_wic_info");

    reg_add_display_wic_info(PM_HARDWARE_FRODO,
                            fellowship_mbrd_display_wic_info,
                            "fellowship_mbrd_display_wic_info");
    . . . snip . . .
}
```

- Step 4** In the platform-independent port adapter function, call the platform-specific routines via the registry service:

```
static void show_pa_detail(uint pa_bay,boolean override)
{
    . . . snip . . .
    reg_invoke_display_wic_info(slotptr->pa_type, pa_bay);
    . . . snip . . .
}
```

7.17.3 For a Platform-Specific Action Implemented on One Platform

When one-to-one mapping is needed or when build modularity is a concern, create a function vector or tables of function vectors. This forces a single implementation when called.

An example of one-to-one mapping is in the DMA engine support code. The hardware DMA engine is used to coalesce a particle-based packet into a contiguous packet on MIPS platforms. The function vector returns the number of the DMA channel to use for the coalesce operation:

```
dma_pak_move_handle_t *dma_pak_move_register (hwidbtype *idb,
                                              int *tx_cnt_ptr)
{
    . . . snip . . .
    hndl->dma_chan_num = dma_eng_alloc_channel(idb);
    . . . snip . . .
}
```

For example, you would follow these steps:

- Step 1** Define an extern function vector in a platform-independent header file. For example, here is a definition from the src-4k/dma_pak_move.h file:

```
typedef ulong (*dma_eng_alloc_channel_t) (hwidbtype * hwidb);
extern dma_eng_alloc_channel_t dma_eng_alloc_channel;
```

- Step 2** Set the global function vector to the default, as in the default implementation of the function vector shown here:

```
ulong dma_eng_alloc_channel_default (hwidbtype * idb)
{
    .......snip...
    return (1)
}
```

- Step 3** Within the platform-specific source, declare the implementation of the function for this platform. For example, in the platform-independent `src-4k/dma_pak_move.c` file:

```
dma_eng_alloc_channel_t dma_eng_alloc_channel =
dma_eng_alloc_channel_default;
```


Interprocessor Communications (IPC) Services

8.1 Introduction

This chapter includes information on the following topics:

- IPC Services: Overview
- Operational Environment
- IPC Communication: Overview
- IPC Terminology
- Port Naming Services
- IPC Message Format
- Unreliable IPC Messaging
- Unreliable IPC Messaging with Notification
- Reliable IPC Messaging
- Remote Procedural Call (RPC) Messaging
- IOS-IPC Process
- IPC Multicast Messages
- Manipulate the Seat Table
- Manipulate the Port Table
- Manipulate the Message Retransmission Table
- Send IPC Messages: Overview
- Receive IPC Messages: Overview
- Simulate RPCs
- Congestion Status Notification Capability
- Write an IPC Application
- Implementing IPCs on the RSP Platform
- Useful Commands for Debugging an IPC Component
- IPC Master
- IPC Get Port Notifications

- References

Note Cisco IOS IPC development questions can be directed to the interest-ipc@cisco.com and ipc-dev-team@cisco.com aliases.

8.2 IPC Services: Overview

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

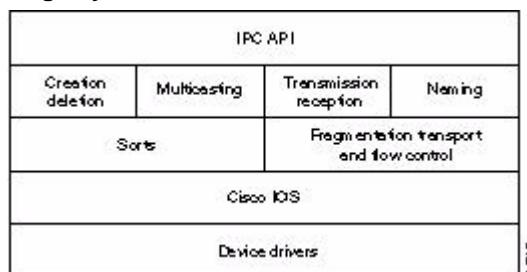
The IPC messaging system provides transparent network and local interprocessor communications. To accomplish this, the following Cisco IOS IPC services are provided:

- 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. Figure 8-1 shows an overview of these interfaces.

Figure 8-1 **IPC Message System Interfaces**



8.3 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.

Table 8-1 IPC Communication Services Environments (continued)

Environment	Description
Loosely coupled	The entities exist on different processors that are not closely related. The communication mechanisms differ significantly from those used 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 includes 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).

8.4 IPC Communication: Overview

All IPC communication takes place between two cooperating entities—a source and a destination—that exchange messages. Communication also happens via the ethernet between processors that can be mentioned in the operational environment. Messages are sent from source ports and received at the destination ports. Source ports are specified so that receiving entities can send return messages to the originator. Ports are identified by port identifiers.

In IPC communication, the users know only about the destination ports and its use in IPC communication. Source port (control port) is used only for internal IPC control communications. At present, the communication is unidirectional. Generally, a port belongs to a single, unique entity that is responsible for processing any messages that arrive on the port.

Multicast communication is enabled in IPC. Using the IPC multicast feature, you can group ports to form a multicast port so that messages can be sent to more than one destination with a single transmission. Multicast ports are groups of ports that can be aggregated and referenced as a single port so that messages can be transmitted from one source to multiple destinations.

Ports are addressed using a port identifier. 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 messages to the destination.

In order for a client to send or receive IPC messages, the client must create local end points or ports and open the destination ports.

When a 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 is 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 name.

8.5 IPC Terminology

Entity

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.

Message

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.

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.)

Port

A *port* is a communications end point. There are two basic types of ports, unicast and multicast. A 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.

Port Identifier

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.

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.

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. A seat uses the platform-specific transport driver to communicate with other seats.

Seat Manager

The *seat manager* is an entity responsible for the local seat. The seat manager is responsible for the following:

1. Assigning port numbers to all its ports.
2. Ensuring that all local port names are unique.
3. Providing seat information to the IPC Master (Zone Manager).
4. Initializing IPC services on the seat.

Zone

A *zone* is a collection of seats between which communications is directly possible.

Zone Manager or IPC Master

The zone manager is referred to as the IPC master in IPC terminology. The IPC Master 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 IPC Master. The IPC Master is also responsible for resolving interseat port names.

In the simplest operational environment, IPC communication is coordinated by the seat manager and the IPC Master. 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 IPC Master is responsible for connecting the seats to a message-routing service.

8.6 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

8.6.1 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.

8.6.2 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

8.6.2.1 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.mgr`.

`viper#1#6:fastsw.mgr`

8.6.2.2 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	IPC Master port for this zone.
IPC Master:Echo	Echo port for this seat.
IPC Master:Control	Control port for this seat.

8.7 IPC Message Format

IPC supports a maximum transmission unit (MTU) of 2^{16} bytes. As a result, an IPC message can be 4096 bytes long. An IPC message comprises of a header followed by a data section. IPC message headers are setup for IPC clients when a message buffer is allocated.

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;
```

Figure 8-2 illustrates the version 0 of the IPC message format.

Figure 8-2 IPC Message Format- Version 0

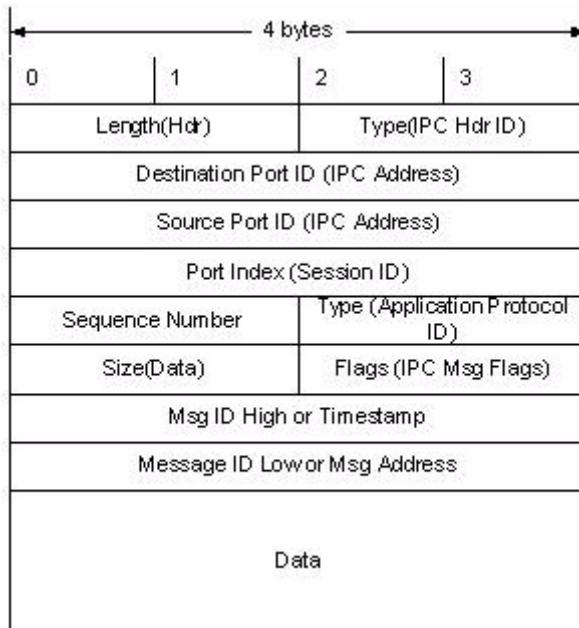
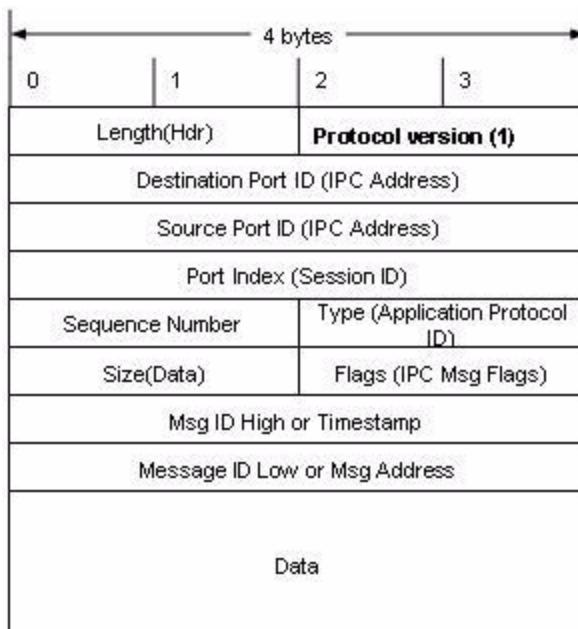


Figure 8-3 illustrates the version 1 of the IPC message format.

Figure 8-3 IPC Message Format-Version 1

The IPC message header begins with a header description word that contains the type of the IPC header and its length. Currently, only one type of IPC header (0) is defined with a length of 32 bytes. The rest, 4064 bytes, is for the data section.

The next two fields of the IPC message header are the IPC addresses for the destination IPC port, and the source IPC port. Next comes the port index field. The port index field is the session ID used by the destination IPC while processing incoming messages.

The next field is the IPC client-specific 16-bit message identifier. This identifier must be specified when the IPC client allocates an IPC message buffer. Next to the message identifier is a sequence number that is used in reliable connection-oriented communication.

The last field is made up of the 16-bit IPC message flag field and a 16-bit data length field. IPC defines various message flags that alter the receive processing of an IPC message.

The 4 message flags related to the type of communication are:

- `IPC_FLAG_DO_NOT_ACK` (Unreliable Message)
- `IPC_FLAG_ACK` (Reliable Message)
- `IPC_FLAG_NACK` (Unreliable with Notification Message)
- `IPC_FLAG_SEQ_NO_ACK` (Unreliable with Notification Message)

The `IPC_FLAG_DO_NOT_ACK` and `IPC_FLAG_SEQ_NO_ACK` flags differentiate between the communication modes. The `IPC_FLAG_DO_NOT_ACK` flag overrides all other IPC message flags and declares a message as an IPC client message. The `IPC_FLAG_SEQ_NO_ACK` flag can alter the acknowledgement processing of the message sequence numbers. When the `IPC_FLAG_SEQ_NO_ACK` flag is set, only negative acknowledgement messages are sent. Otherwise, acknowledgement messages are sent during the acknowledgement processing. The `IPC_FLAG_ACK` and `IPC_FLAG_NACK` flags are used in internal messages dedicated for the purpose of acknowledging the state of the communication session.

The message flags related to control processing, or RPC messaging are:

- `IPC_FLAG_CONTROL`

- `IPC_FLAG_IS_RPC_REPLY`

The `IPC_FLAG_CONTROL` flag identifies the IPC message as being destined for a control IPC port. Control IPC ports have their own IPC address, so this flag is mostly informational. The `IPC_FLAG_BOOTSTRAP` flag identifies platform-dependent IPC messages related to IPC processing. These messages tend to deal with IPC discovery and the passing of the seat state. The `IPC_FLAG_IS_RPC` and `IPC_FLAG_IS_RPC_REPLY` flags are related to RPC processing. When IPC messages set with these flags are received, they are passed to the RPC processor.

The following 2 message flags are used in IPC fragmentation processing:

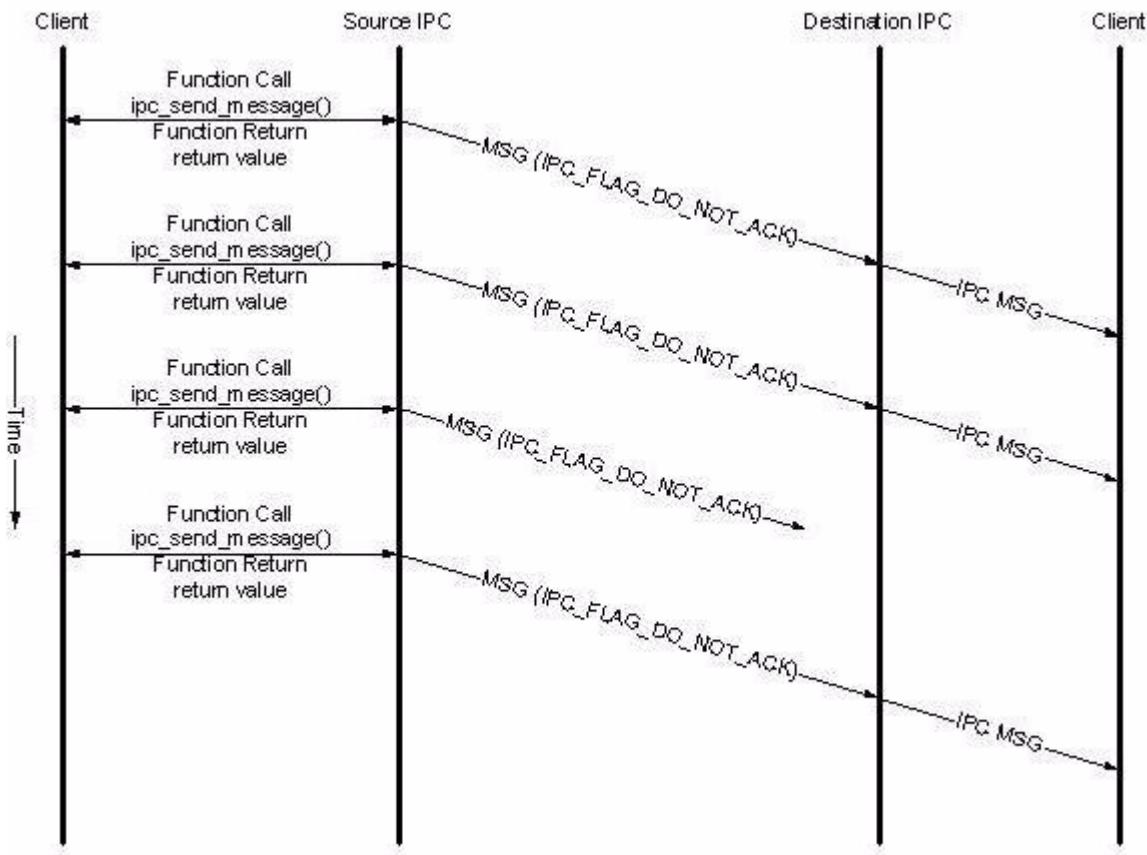
- `IPC_FLAG_IS_FRAGMENT`
- `IPC_FLAG_LAST_FRAGMENT`

The data length field in the IPC message header describes how large the data section is in bytes.

8.8 Unreliable IPC Messaging

IPC clients can use either the `ipc_send_message()` call or the `ipc_send_message_blocked()` call for reliable (acknowledged) communication. The difference is that `ipc_send_message()` is a non-blocked call and `ipc_send_message_blocked()` is a blocked call. However, if you use the `ipc_send_message_blocked()` call, there will be no acknowledgement message and the application will timeout.

Figure 8-4 shows a non-blocking send for an unreliable IPC message.

Figure 8-4 Non-Blocking Send for Unreliable IPC Message

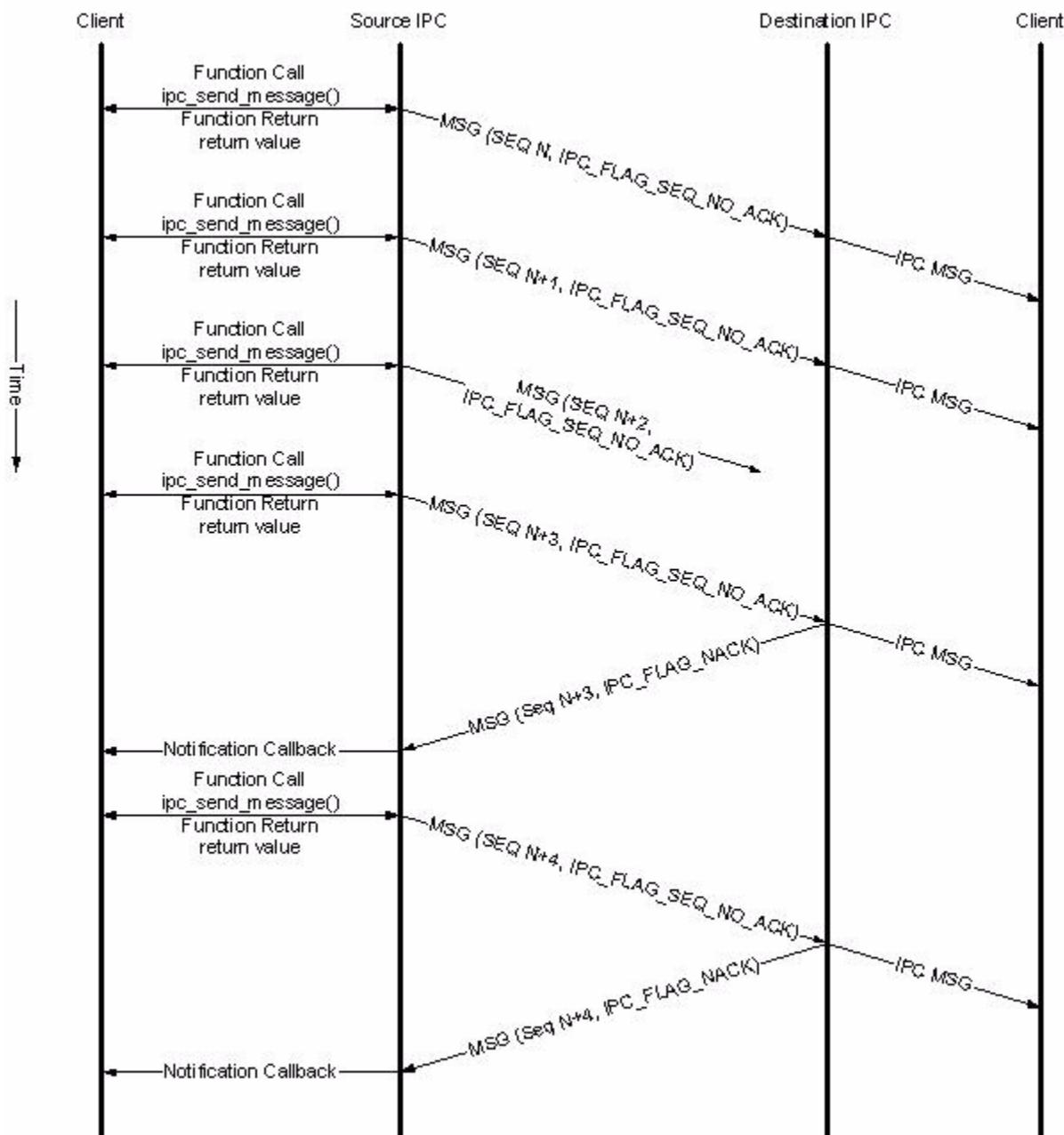
When an unreliable IPC message is transmitted, the `IPC_FLAG_DO_NOT_ACK` bit is set in the IPC message flags field of the IPC message header. The setting of this indicates to the destination IPC seat manager to bypass all other receive processing and to pass the message to the receiving IPC client using the desired receive method.

8.9 Unreliable IPC Messaging with Notification

IPC clients use the unacknowledged send, `ipc_send_message()`, for unreliable IPC communication with notification. If an IPC client uses the acknowledged send, `ipc_send_message_blocked()`, the execution of that message is indefinitely blocked because unreliable communication with notification does not support acknowledgements.

Figure 8-5 shows a non-blocking send for unreliable with notification IPC message.

Figure 8-5 Non-Blocking Send for Unreliable with Notification IPC Message



When an unreliable IPC message with notification is sent, the `IPC_FLAG_SEQ_NO_ACK` bit flag is set in the IPC message flag field of the IPC message header. The sequence number of the current communication session is sent in the sequence number field of the IPC message header. The receive side will respond when the sent sequence number and the received sequence number is out of sync. There is no mechanism to regain the synchronization. The notification callback to the IPC client does not contain any information about lost, or last received messages.

8.10 Reliable IPC Messaging

IPC clients can use reliable IPC communication with either the acknowledged send, `ipc_send_message_blocked()` or the unacknowledged send, `ipc_send_message()`. The sequence numbers used in reliable communication are unique to an IPC session. IPC sessions are uni-directional, or simplex communication pipes.

Figure 8-6 Blocking Send for Reliable IPC Message

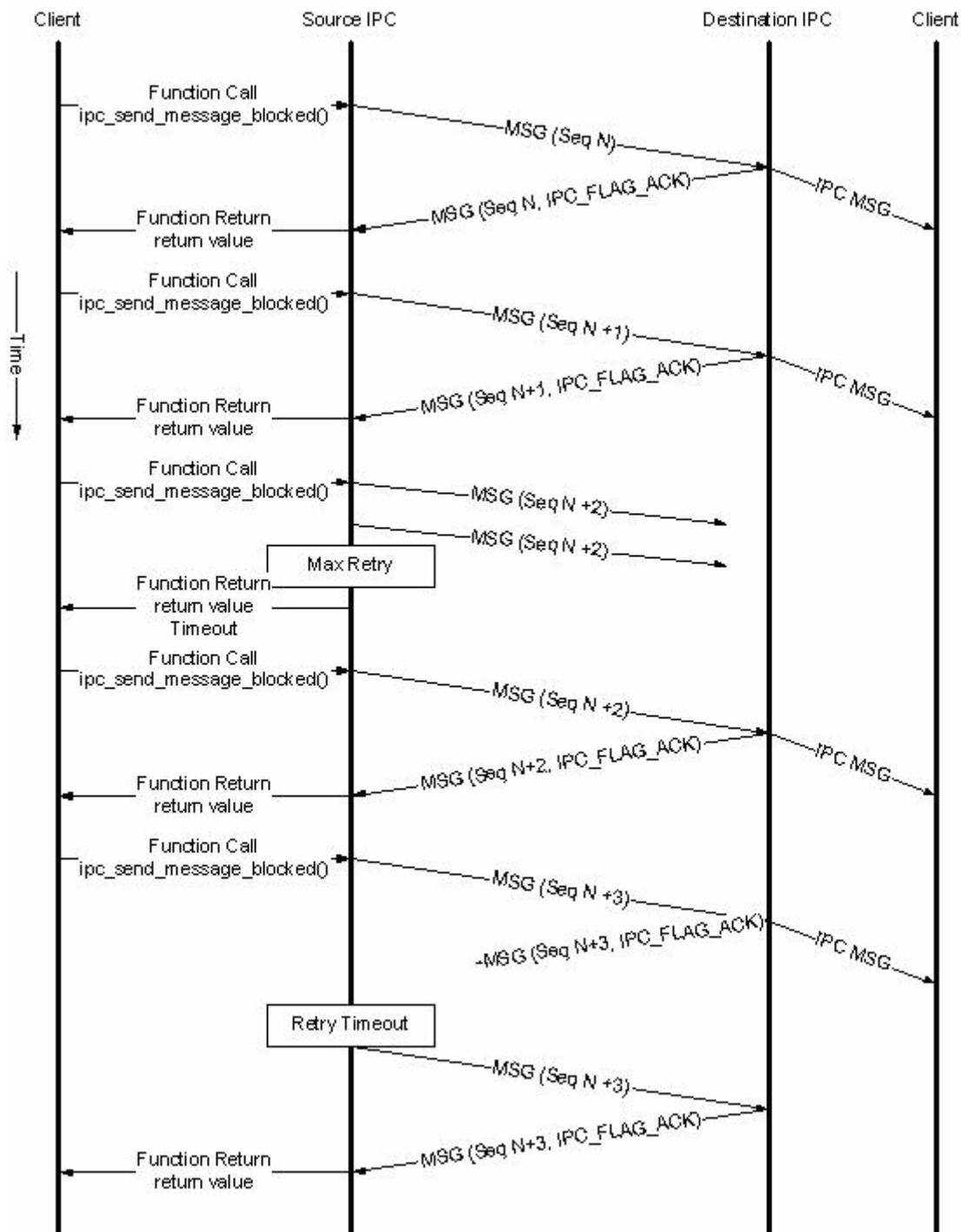


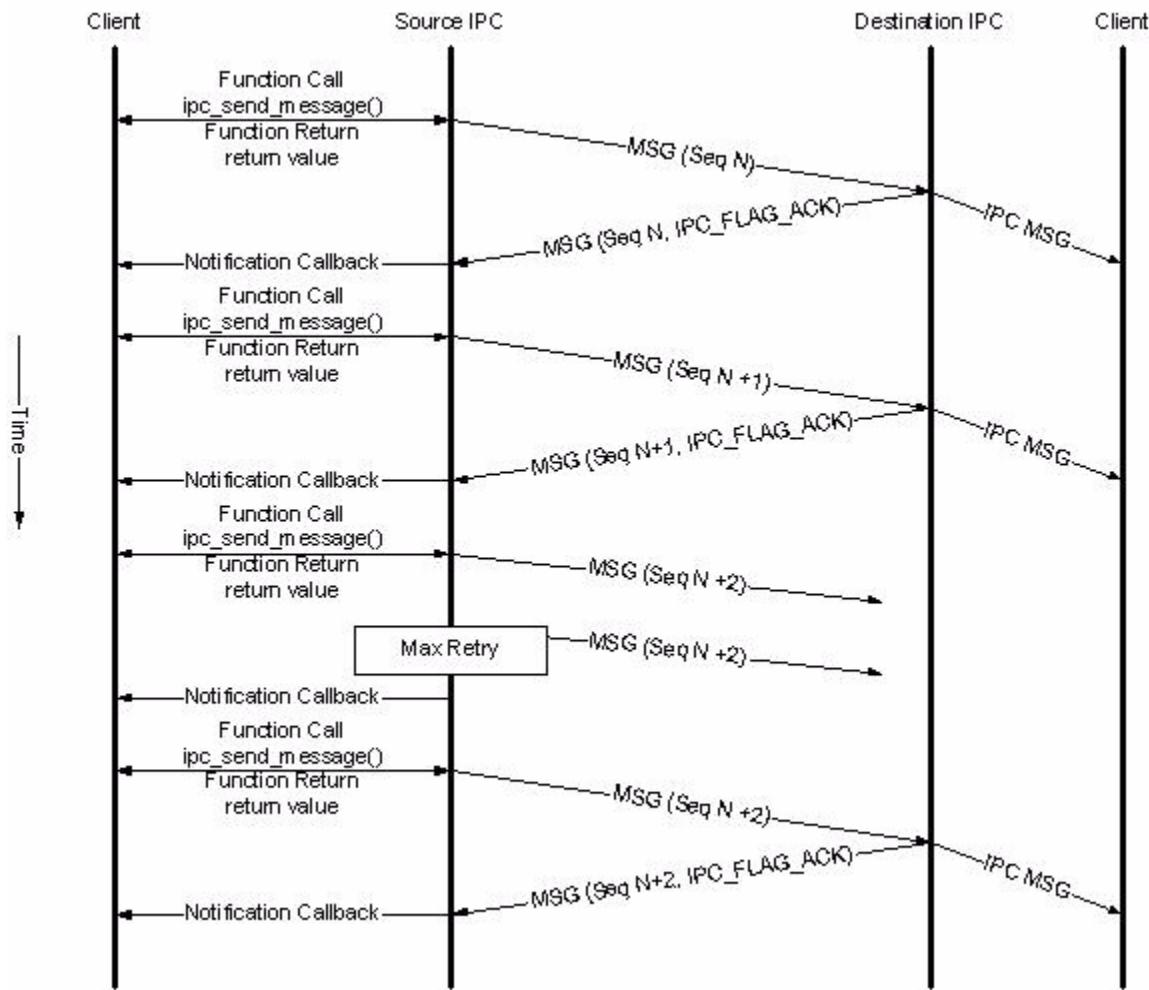
Figure 8-6 shows a blocking send for a reliable IPC message.

In reliable IPC messaging, `IPC_ACK` will be copied with the `msgID_high` and `msgID_low` sequence fields that is used by the sender to find the IPC message that is sent.

When a reliable IPC message is sent, the sequence number field in the IPC message header is set by the sending IPC and checked by the receiving IPC seat manager. For each successfully received message, the receiving IPC seat manager sends an acknowledgement back to the sender. An acknowledgement is an IPC message with the `IPC_FLAG_ACK` bit set in the IPC message header. Also, the Sequence field and the Index Port field in the IPC message header are copied from the IPC message being acknowledged. If a message fails to reach its destination, the sending IPC will timeout and notify the sending IPC client of the failure to send. The IPC seat manager of the sender attempts to re-send the message periodically up to the maximum number of retries set by the platform. After which, it notifies the IPC client of the failure to send the IPC message.

When a reliable IPC message is sent using the acknowledged send, `ipc_send_message_blocked()`, the IPC client sending the message suspends work until an acknowledgement for the message is received, or a timeout indicating the message has failed to reach its destination is received. In either case, the return value of the `ipc_send_message_blocked()` function will indicate the status of the IPC message.

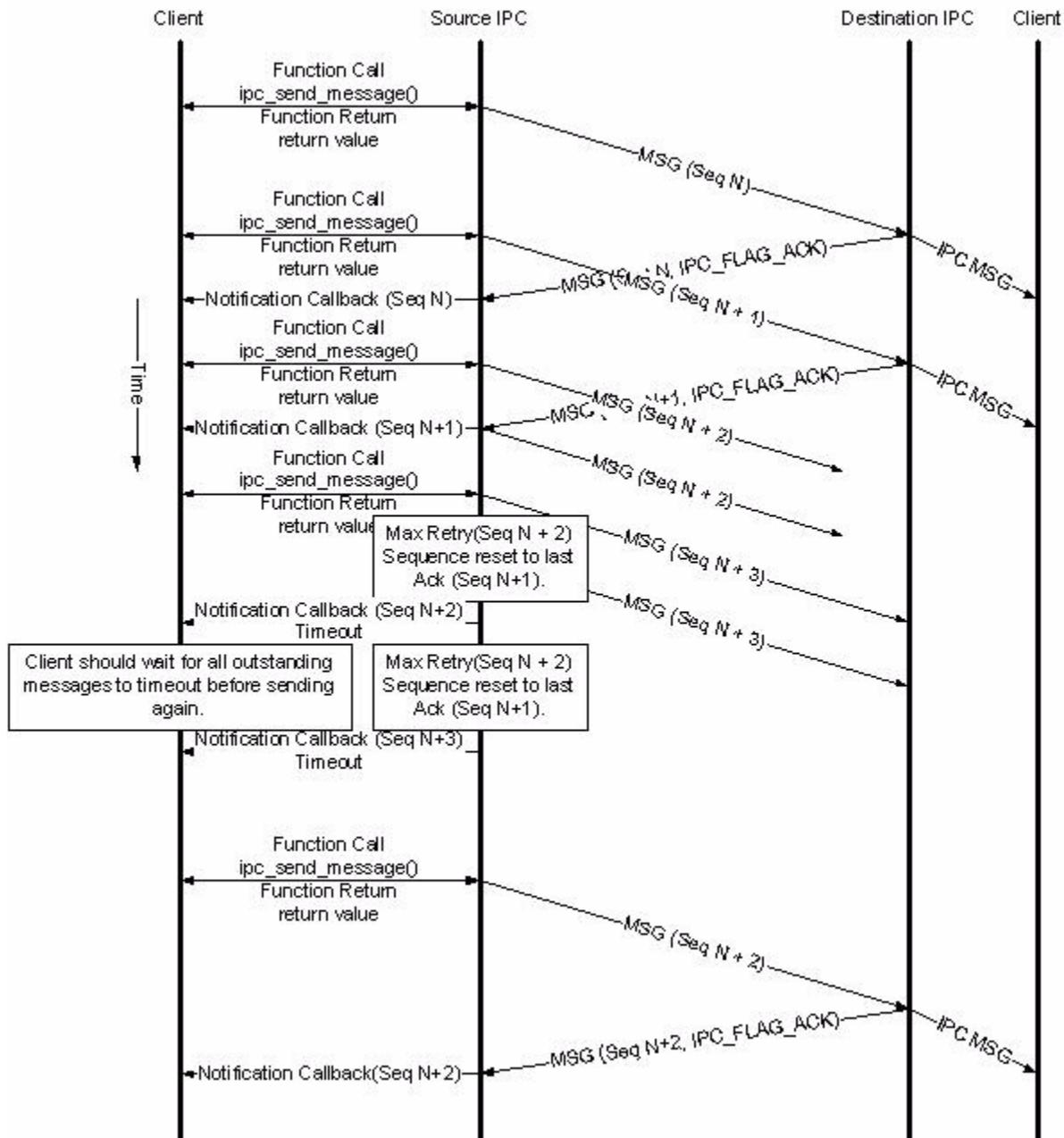
Figure 8-7 shows a non-blocking send for a reliable IPC message without overlapping sends.

Figure 8-7 Non-Blocking Send for Reliable IPC Message without Overlapping Sends

When a reliable IPC message is sent using the unacknowledged send, `ipc_send_message()`, the IPC client is allowed to execute after handing the message to IPC for sending. The IPC client is notified of the final status of a sent IPC message using the notification callback.

Figure 8-8 shows a non-blocking send for a reliable IPC message with overlapping sends.

Figure 8-8 Non-Blocking Send for Reliable IPC Message with Overlapping Sends



When an IPC client sends unacknowledged messages in rapid succession before receiving acknowledgements, the IPC client is responsible for waiting for the IPC to recover any lost, or out of sequence IPC messages before sending more IPC messages.

The transmitting IPC seat resets the IPC session sequence number to the last acknowledged number when an IPC message send fails. If there are three outstanding messages, N, N+1, and N+2, when N reaches the maximum number of retries, IPC resets the IPC session sequence number to N. So, if an IPC client sends another three messages after receiving a timeout, those sequence numbers will be N, N+1, and N+2, even though the N+1 and N+2 messages are still pending.

8.11 Remote Procedural Call (RPC) Messaging

Figure 8-9 Blocking Send for RPC Messages

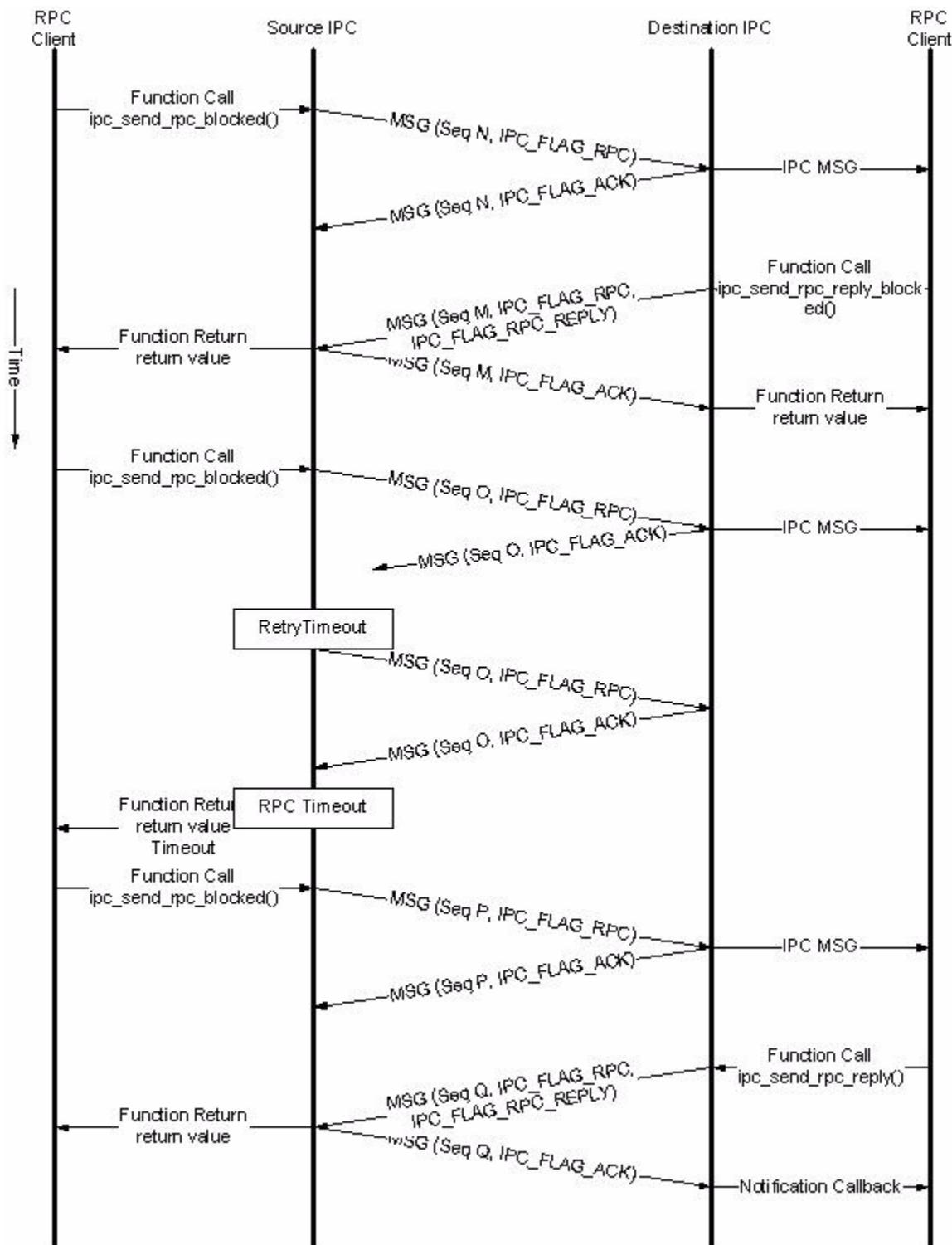


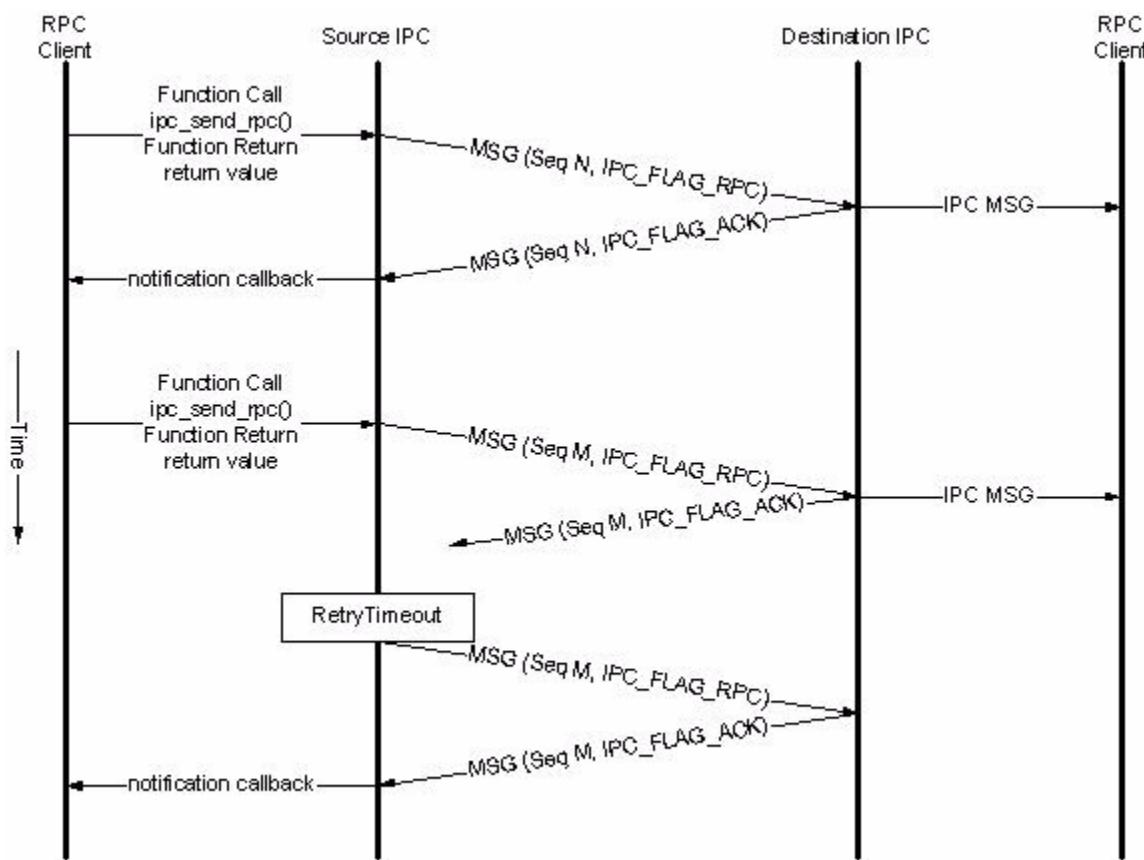
Figure 8-9 shows a blocking send for RPC messages.

The RPC communication service uses a modified form of the reliable connection-oriented communication service. RPC sends are performed on an open reliable messaging IPC session. RPC sends can be done on unreliable IPC sessions, but it is strongly recommended not to do so. The sequence numbers used by the RPC messaging are local to the IPC seat rather than the IPC port.

In RPC messaging, the remote client does not open sessions for sending RPC replies. IPC uses the internal IPC control sessions to send RPC replies to the sender. Delivering RPC replies to the sender client is the responsibility of IPC.

In the example shown in Figure 8-10, the RPC client opens a reliable IPC session with the remote RPC client prior to performing an RPC send. Likewise, the remote RPC client opens another reliable IPC session to send the RPC reply. Both sides close the IPC sessions on the completion of messaging.

Figure 8-10 Non-Blocking Send for RPC Messaging



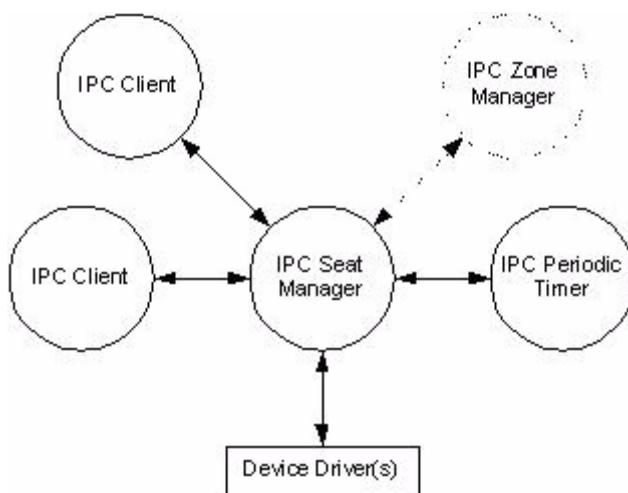
Most IPC clients use the blocking send when transmitting an RPC request. RPC replies can also be sent using the blocking and non-blocking send. RPC replies uses the IPC control ports. There is no need for the clients to open a separate session for this.

RPC provides a Non-Blocking form of the RPC send, `ipc_send_rpc()`. RPC clients are responsible for receiving RPC replies when they use the non-blocking RPC send. However, the `ipc_send_rpc()` function is not commonly used.

8.12 IOS-IPC Process

Figure 8-11 illustrates the IOS-IPC process.

Figure 8-11 IOS-IPC Process



8.13 IPC Multicast Messages

IPC multicast provides the capability to transmit a message to multiple ports on multiple seats. IPC makes use of hardware multicasting when a medium contains the capability of multicasting. If a medium does not have the multicasting capability, IPC emulates multicasting in its layer. By providing the multicast capability at the IPC layer, applications become less complex. IPC multicast helps improve the performance of IPC clients, such as ICC and XDR, that use multicast emulation.

8.13.1 Multi-Operating System Support

IPC multicast works in IOS, ION, LINUX, and any other operating system that is POSIX compliant (ION, QNX, LINUX, and so on.). The IPC protocol has been modularized and redesigned to run in a preemptive operating system. Because IPC multicast is built on top of the core IPC, it uses the underlying redesigned architecture. No special changes are required for IPC multicast.

In a multi-OS IPC, the client process communicates with the local IPC server, and the local IPC server interacts with other remote IPC servers. This requires the IPC multicast code to be divided into two sets:

- An IPC client library that allows communication between the client process and the local IPC server process.
- An IPC server process that is responsible for communication with remote servers as well with its local clients.

A client process and the local IPC server communicate with each other using the system message queues. For example, a client process creates a multicast port to receive messages. This request is communicated to the local IPC server process. The port is associated with the client process through a process ID. When an IPC message arrives for the port associated with the client process, the

message is delivered using a queue that is shared between the processes. Similarly, the client that is sending the message is associated with a port or a group during the open operation. Both the client and the local server process can access the message buffers.

In Cisco IOS, a watched queue performs better than local sends. IPC multicast focuses on sending messages to ports on different nodes.

8.13.2 Asynchronous Control Messaging

IPC unicast uses synchronous (blocking) calls for control messages. IPC multicast uses the same model for a consistent user behavior, but also supports asynchronous control messaging (non-blocking) to allow application flexibility. The advantages of asynchronous and synchronous messaging are described in this section.

- Advantages of asynchronous control messages:
 - Scalability and performance—can process several operations simultaneously.
 - Ease of application use—process (or thread) need not be blocked.
 - Standards-based.
- Advantages of synchronous control messages:
 - Fewer timing related bugs.

8.13.3 Functional Structure

A reliable IPC multicast transport protocol consists of the various protocol components such as:

- Group membership management
This includes the enrollment of receivers to a specified multicast group and query operations.
- Connection management
This includes creating and closing a session and data transfer functions.
- Reliability management
This includes detecting the need to request a retransmission, requesting the retransmission, and retransmitting the requested messages.
- Buffer management
This includes the allocation of IPC messages and resource accounting.
- Flow or congestion control
Flow control deals with the availability of buffer space at the receivers and congestion control that regulates the traffic that enters the transport driver.

8.13.4 System Flow

This sections describes the various system flows like, control messages, IPC-platform interaction, IPC-client interaction, IPC-ISSU interaction, and message flow.

8.13.4.1 Control Messages

The following are the list of control messages:

- IPC_TYPE_MCAST_GROUP_CREATE_REQUEST
- IPC_TYPE_MCAST_GROUP_CREATE_RESPONSE
- IPC_TYPE_MCAST_GROUP_LOCATE_REQUEST
- IPC_TYPE_MCAST_GROUP_LOCATE_RESPONSE
- IPC_TYPE_MCAST_GROUP_DELETE_REQUEST
- IPC_TYPE_MCAST_GROUP_OPEN_REQUEST
- IPC_TYPE_MCAST_GROUP_OPEN_RESPONSE
- IPC_TYPE_MCAST_GROUP_CLOSE_REQUEST
- IPC_TYPE_MCAST_GROUP_BIRTH_NOTIFICATION
- IPC_TYPE_MCAST_GROUP_DEATH_NOTIFICATION
- IPC_TYPE_MCAST_GROUP_MEMBER_ADD_REQUEST
- IPC_TYPE_MCAST_GROUP_MEMBER_ADD_RESPONSE
- IPC_TYPE_MCAST_GROUP_MEMBER_REMOVE_REQUEST
- IPC_TYPE_MCAST_GROUP_MEMBER_REMOVE_RESPONSE
- IPC_TYPE_MCAST_GROUP_MEMBER_JOIN_NOTIFICATION
- IPC_TYPE_MCAST_GROUP_MEMBER_LEAVE_NOTIFICATION
- IPC_TYPE_MCAST_APPCLASS_MEMBER_JOIN_NOTIFICATION
- IPC_TYPE_MCAST_APPCLASS_MEMBER_LEAVE_NOTIFICATION
- IPC_TYPE_MCAST_APPCLASS_SUBSCRIBE_REQUEST
- IPC_TYPE_MCAST_APPCLASS_SUBSCRIBE_RESPONSE

8.13.4.2 IPC-Platform Interaction

8.13.4.2.1 Initialization

A platform needs to provide a medium that supports hardware multicast. During seat creation, the platform denotes if the seat can be reached using such a medium:

```
ipc_error_code ipc_reachable_via_hardware_multicast (ipc_seat seat_id,  
boolean reachable)
```

The platform uses the same output vector to transmit multicast messages as unicast messages and distinguishes between the two types of messages based on the destination ID. For unicast, the destination ID is a seat ID and for multicast, it is the group ID.

8.13.4.2.2 Driver Operations

IPC is a transport-independent protocol, which allows platforms to run IPC on different mediums—Ethernet, UDP, and shared memory.

When a group is created, the group ID must be mapped to the hardware address. Some platforms have a limitation on the number of multicast addresses it can support. Similarly, when a group is removed the hardware address must be disassociated from the platform.

IPC multicast sends messages to a set of seats, where the set is determined by the group membership. The hardware multicast address must be updated with each add and remove group member operation. IPC uses the seat ID of the port that is added or removed to inform the platform.

When a group is created, on the receiver side, the joining member should listen to the specific hardware address. Similarly, the member should stop listening to the hardware address when leaving the group.

8.13.5 IPC Multicast Message Format

To identify a multicast message, IPC checks the fields of the flags in the message header. The `IPC_FLAG_MULTICAST` flag is set for all multicast messages. Similar to the unicast model, the type of service of the multicast message is determined by the message header flags, `IPC_FLAG_RELIABLE` and `IPC_FLAG_DO_NOT_ACK`. The group ID is stored in the `dest_port` field of the flags.

When the `IPC_FLAG_CONTROL` flag is set along with `IPC_FLAG_MULTICAST`, it indicates a broadcast control message for the group.

In the following example, the multicast message is displayed in bold format.

```
#define IPC_FLAG_CONTROL      0x0200 /* This is a control message */
#define IPC_FLAG_RELIABLE     0x0400 /* This is a RELIABLE IPC message */
#define IPC_FLAG_SEQ_SYNC     0x0800 /* Sync seq # in seq window */
#define IPC_FLAG_ACK_IMMEDIATE 0x1000 /* used by the sender to force rx ACK */
*/
#define IPC_FLAG_ISSUMSG      0x2000 /* This is an ISSU negotiation
message*/
#define IPC_FLAG_MULTICAST    0x4000 /* This is a multicast message */
```

8.13.6 IPC Message Type Definition

The IPC message type definition (message descriptor used by IPC clients) is modified to meet the needs of multicast transmission. The receiver set is represented by a bit vector, which is `ulonglong`. The maximum number of receivers in a group is restricted to 64. In the bit vector, if a particular bit position is set to `i`, then that receiver is yet to acknowledge the message. For multicast, IPC has to wait for acknowledgements from all the receivers. As each ACK arrives, the bit vector is updated. Only when the bit vector is empty, the message is considered to be successfully sent. Similarly, the `rpc_bitfield` flag is set for RPC messages. As responses arrive the `rpc_bitfield` flag is removed from the bit representing the member.

In the following example, the IPC multicast message type definition is highlighted in bold format:

```
struct ipc_message_ {
    thread_linkage links;

    /*
     * The below pointer is for chaining up the message onto the
     * holdq. See IPC_HOLDQ_XXX macro below.
     */
    void *holdq_link;

    ipc_message_header *header;           /* Pointer to the message header */
    void *data;                          /* Pointer to the message data */
    ipc_port_data *port_p;               /* Pointer to the received dest port */
}
ipc_port_info *dest_port_info;          /* Pointer to port info from open */
void *data_buffer;                     /* Pointer to the buffer that holds
the data */

/*
```

Manipulate the Seat Table

```

        * For use by server:
        */
sys_timestamp retry_time;           /* Time at which we retransmit */
ulong retry_period;                 /* Time between retries */
mgd_timer rpc_timer;               /* Managed timer used by rpc */
uint rpc_timeout;                  /* rpc timeout value in seconds */
volatile uint32 flags;              /* Message status */
watched_boolean *not_blocked;       /* Boolean to block the message */
ipc_error_code result;              /* What happened to this message */
uint sequence;                     /* Unique sequence number of this
                                         message [TABLE KEY] */

        int retries;                      /* How many times we've resent this */

*/
ipc_free_func_t ipc_free_func;       /* Function to free data buffer */
void *rpc_result;                  /* Store RPC result here */
mgd_timer issu_timer;               /* managed timer used by IPC ISSU*/

        ipc_callback notify_callback;     /* Callback routine for message */
void *notify_context;               /* Context for message callback */

        ipc_fragment_t *fragment;
ushort refcount;

/*
 * Bundling - submessages in a bundled one
 */
queueuetype submsgQ;

/*
 * Multicast
 */
ulonglong ack_mcast_bitfield;
ulonglong ack_ucast_bitfield;
ulonglong rpc_bitfield;
ulonglong failed_bitfield;
ulonglong success_bitfield;
};

}

```

The handling of RPC request-response in unicast and multicast vary. For unicast, the RPC response is attached to the RPC request (via `msg->rpc_result`). Then, the original RPC request is sent back to the sending client. For multicast, the `msg->rpc_result` is not used. The RPC responses are placed in the `msg->rpc_resultQ`. Also, the `msg->recv_set` bit vector is reset to its original value when placed in the `ipc_rpc_req_table`. As RPC responses arrive, the `msg->recv_set` bit vector is updated. When the bit vector is empty, the client is notified that all the responses have arrived.

8.14 Manipulate the Seat Table

The following sections describe manipulating the seat table:

- Seat Table: Description
- Create a Seat
- Get Information about a Seat
- Reset a Seat

8.14.1 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 32-bit (uses uint as datatype) 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 heard from this seat */
}
ipc_sequence last_ack; /* Last ack sequence number rcv'd */
};
```

8.14.2 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
```

8.14.3 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);
```

8.14.4 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);
```

8.15 Manipulate the Port Table

The following sections describe manipulating the port table:

- Port Table: Description
- Create a Port
- Register a Port
- Open a Port
- Find a Port
- Close a Port
- Remove a Port

8.15.1 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];
};
```

8.15.2 Create a Port

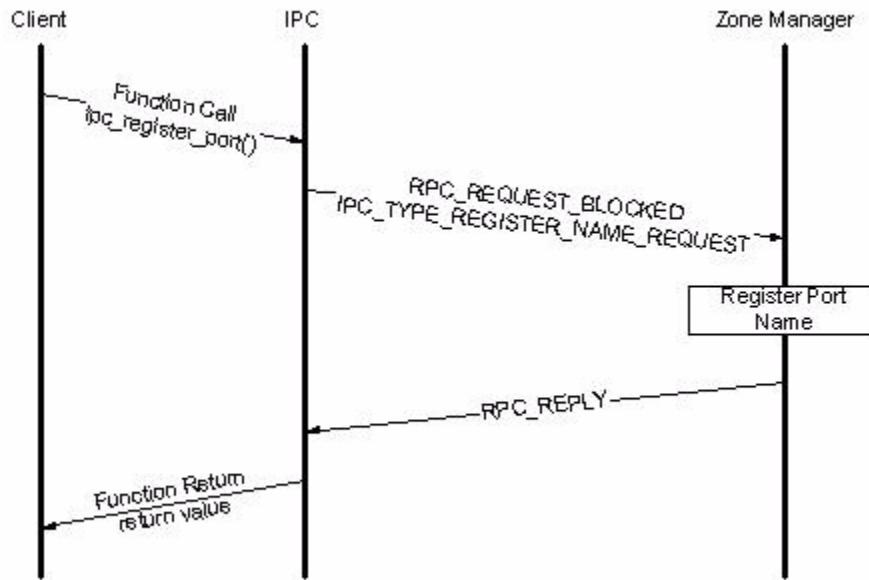
IPC clients can create an IPC port by using the `ipc_create_named_port()` function to communicate with each other. While creating an IPC port, the IPC client must specify the port name, whether or not the new port is local to the IPC seat, and whether or not it is used for uni-cast communication. IPC does not support multi-cast communication. The IPC client should also specify the preferred receive scheme to be used for the new IPC port.

An IPC client may receive IPC messages from a watched message queue, a callback function called from the IPC seat manager process, or a callback function called from the IPC receive handler. Because of the implementation of IPC, the IPC receive handler may actually be called from the process rather than in interrupt. The IPC client can specify the receive context and the pointer to a data block that will be passed to the callback function each time it is called to process a received IPC message. The receive context pointer of an IPC port will be set at the IPC port creation.

On the successful creation of an IPC port, a portID is returned to the IPC client. This portID is used in all correspondence through IPC. After receiving a portID, the IPC client registers the IPC port with the global name service by using the `ipc_register_port()` function. Registration of the port is mandatory. It should be done by the IPC slaves to register themselves with the IPC Master for distributed communication. Registration of the port is mandatory. This registration allows other remote IPC clients to be aware of the newly created IPC port.

Figure 8-12 shows the name registration message flow.

Figure 8-12 Name Registration Message Flow



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);
```

8.15.3 Register a Port

To register a port by its port identifier, you can use the [ipc_register_port\(\)](#) function.

```
ipc_error_code ipc_register_port(ipc_port_id port);
```

8.15.4 Open a Port

An IPC client with its own IPC port can establish an uni-directional communication session with another IPC client by either using the destination port's IPC address or the IPC port name. To open a port you can use the [ipc_open_port\(\)](#) and [ipc_open_port_by_name\(\)](#) functions.

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);
```

To know about the port events, clients can set the [tx_port_event_notify_callback\(\)](#) function

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_CALLBACK`.
- Notify context—This field contains an optional context for the callback `ack` routine to use with `IPC_PORT_CALLBACK`.

An IPC client needs to fill in the notify callback and notify context fields if the `IPC_PORT_CALLBACK` flag is set in the port features fields, otherwise these fields will be set to NULL. The notify fields can be set prior to the `ipc_open_port()`, or `ipc_open_port_by_name()` function call.

The `IPC_PORT_FEAT_RELIABLE` flag should be set for IPC to use the reliable messaging service. If the unreliable messaging without notification service is used, the `IPC_PORT_FEAT_UNREL` flag should be set. If the unreliable messaging with notification service is desired, the `IPC_PORT_FEAT_UNREL_NOT` flag should be set. The `IPC_PORT_FEAT_RELIABLE`, `IPC_PORT_FEAT_UNREL`, and `IPC_PORT_FEAT_UNREL_NOT` flags are mutually exclusive and one of them is mandatory.

The `IPC_PORT_FEAT_CONTROL` flag is an internal IPC port flag and not for use by IPC clients.

The `ipc_port_info` structure is one of the parameters passed to the `ipc_open_port()` and `ipc_open_port_by_name()` functions by reference. The port features field is the only field that must be initialized when either of the routines are called. If a notification callback is desired, the notify callback and notify context fields should be filled. Fields in bold are initialized by the `ipc_open_port_by_name()`, or `ipc_open_port()` function.

The `ipc_port_info` structure is given below:

```
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 */
```

Manipulate the Port Table

```
ipc_sequencelast_sent; /* Last sequence number assigned */
ipc_sequencelast_heard; /* Last valid sequence number heard
from this seat */
ipc_sequencelast_ack; /* Last ack sequence number rcv'd */

ushort port_index;
uint xmit_pending; /* Number of retransmit msgs */
};

/*
 * port_features bits
 */
#define IPC_PORT_FEAT_RELIABLE 0x00000001/* Reliable transport */
#define IPC_PORT_FEAT_UNREL 0x00000002/* Unreliable transport */
#define IPC_PORT_FEAT_UNREL_NOT 0x00000004/* Unreliable with notification */
#define IPC_PORT_FEAT_CONTROL 0x00000008/* Control port */
#define IPC_PORT_CALLBACK 0x00000010/* For async sends */
```

If a port name is used to open a port and the port name can not be resolved locally on an IPC seat, the seat sends a control message and a name lookup request to the IPC Master. The IPC Master looks up the name in the global name database and sends a response.

Figure 8-13 shows the IPC lookup message.

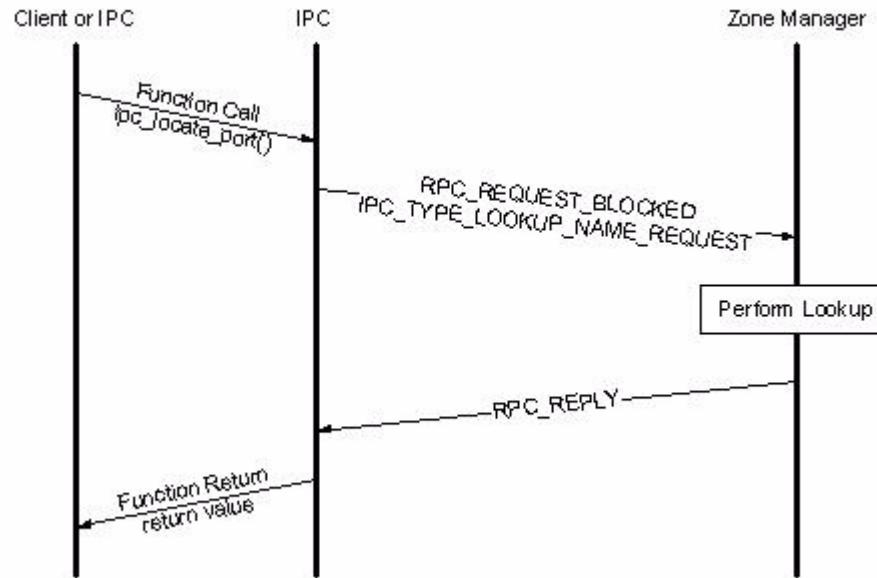
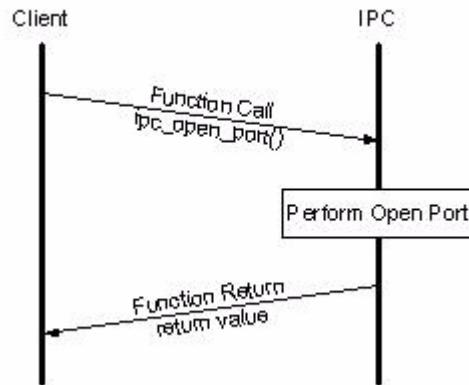
Figure 8-13 IPC Lookup Message**Figure 8-14 Open Port Message Flow for Local Port**

Figure 8-14 shows the open port message flow for a local port.

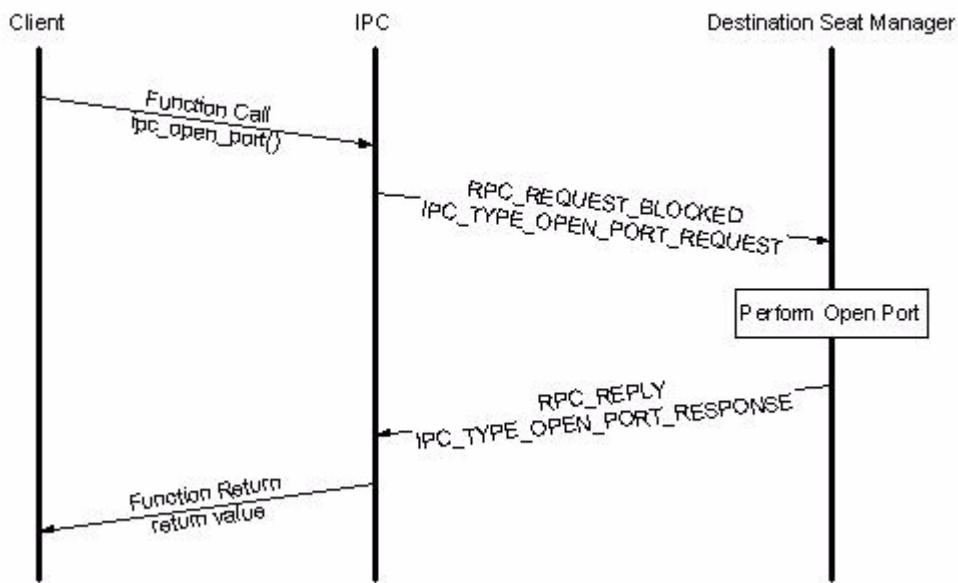
Figure 8-15 Open Port Message for Remote Port

Figure 8-15 shows an open port message for a remote port.

On receiving a positive acknowledgment, the IPC client will have an open port and begin to send messages to the destination port.

An example of the IPC Open Port Request and Response Message Definition:

```

typedef struct ipc_open_port_request_ {
    ipc_port_id    port_id;      /* port to open (IPC Address) */
} ipc_open_port_request;

typedef struct ipc_open_port_response_ {
    ushort        port_index; /* IPC Session ID */
    ushort        error_code; /* result code */
} ipc_open_port_response;
  
```

Notification of the type of reliability to use for a given IPC session is not negotiated when an IPC session is established, but when an IPC message is sent or received. When an IPC seat receives an IPC message, the IPC message flags, `IPC_FLAG_DO_NOT_ACK`, `IPC_FLAG_ACK`, `IPC_FLAG_NACK`, and `IPC_FLAG_SEQ_NO_ACK`, indicate the type of reliability that should be used for the received message.

8.15.5 Find a Port

To locate a port by name, use the `ipc_locate_port()` function.

```
ipc_port_id ipc_locate_port(ipc_name *name);
```

8.15.6 Close a Port

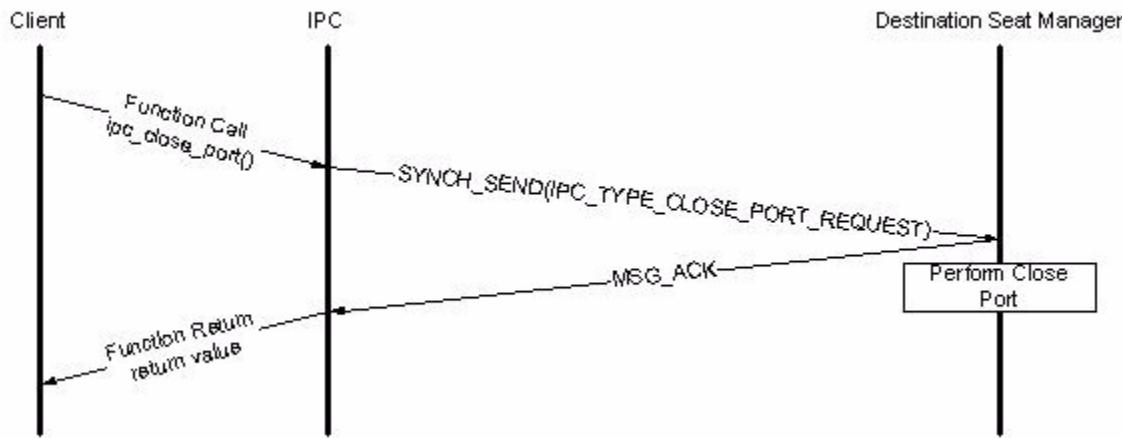
The IPC client is responsible for closing the IPC session. IPC sends a control message, `CLOSE_PORT_REQUEST`, to the IPC seat manager responsible for the end point of the IPC session. The destination IPC seat manager updates itself and the destination IPC port to the status of the IPC session. If the IPC session is with a local IPC port, the closing of the port is handled by the `ipc_close_port()` function.

Figure 8-16 shows a close port message flow.

To close a port, use the `ipc_close_port()` function.

```
ipc_error_code ipc_close_port(ipc_port_info *port_info);
```

Figure 8-16 Close Port Message Flow



If any re-transmissions on a reliable IPC session are pending when a port is closed, IPC defers the closing of the port and handles the closing in the background. However to IPC clients, the IPC session is closed.

An IPC close port request message definition:

```
typedef struct ipc_close_port_request_ {
    ipc_port_id    port_id; /* port to close (IPC ADDRESS) */
    ushort         port_index; /* IPC Session ID */
} ipc_close_port_request;
```

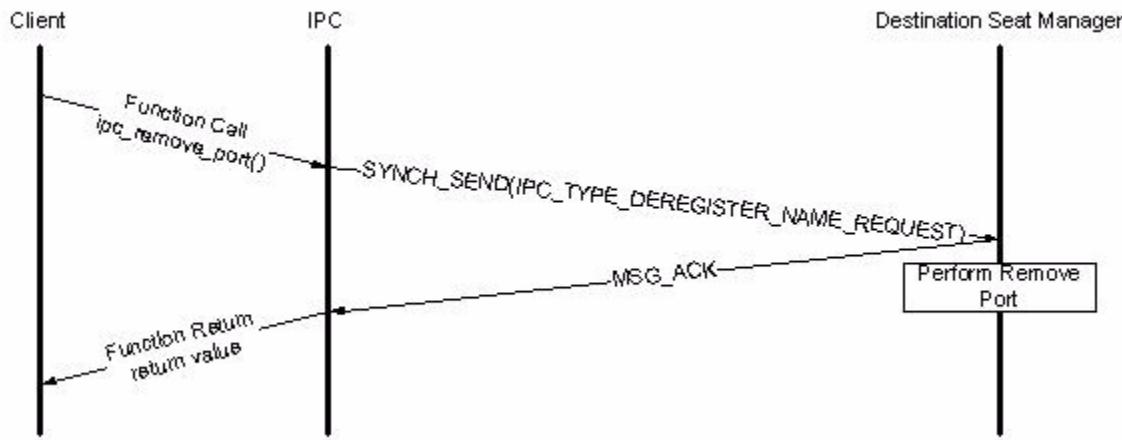
8.15.7 Remove a Port

To remove an IPC port, call the `ipc_remove_port()` function. The `ipc_remove_port()` function releases the memory associated with an IPC port. Before removing a port, ensure that the IPC client has closed all IPC sessions on the IPC port, and there are no pending messages.

Figure 8-17 shows a remove IPC port message flow.

To remove a port and to deregister the port, use the `ipc_remove_port()` function.

```
ipc_error_code ipc_remove_port(ipc_port_id port);
```

Figure 8-17 Remove IPC Port Message Flow

8.16 Manipulate the Message Retransmission Table

The following section describes manipulating the message retransmission table:

- Message Retransmission Table: Description

8.16.1 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. During message retransmission, the sequence number, msgID high, and msgID low are used from the acknowledged message to fetch the original message.

Message retransmission table entries contain the following information:

- Retransmission information
 - Sequence number
 - Retransmit timer
 - Packet pointer
- Notification
 - Callback function
 - Callback context

8.17 Send IPC Messages: Overview

When sending an IPC message, an IPC client must first request for an IPC message buffer from IPC by using the `ipc_get_pak_message()` function. The `ipc_get_pak_message()` function will allocate a buffer for the client and fill the header section of the IPC message. The IPC client can then proceed to fill the data section of the message buffer.

Once the IPC message is ready, the IPC client sends the message by using an acknowledged send, `ipc_send_message_blocked()`, or by using an unacknowledged send, `ipc_send_message()`. For unreliable sends, you can use only the `ipc_send_message()` call. For reliable message sends, you can use either the `ipc_send_message_blocked()` or the `ipc_send_message()` calls. The `ipc_send_message_blocked()` call blocks the application until the acknowledgement arrives or a timeout occurs. The `ipc_send_message()` call allows the application not to block.

An acknowledged send forces an IPC client to suspend work until the receiving IPC seat manager acknowledges the receipt of the sent message, or a timeout occurs on the local seat manager waiting for an acknowledgement. IPC does not restrict the communication services for which an acknowledged send may occur. If an acknowledged send is done on an unreliable connection, the function will not return until a timeout occurs.

An unacknowledged send allows an IPC client to resume execution after passing the IPC message to IPC. For reliable connections, or unreliable connections with notification, if a notification callback is set, the callback is called when IPC receives a successful acknowledgement, a negative acknowledgement, or a timeout happens. Notification callbacks are setup when a port is opened for communication.

8.17.1 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);
```

8.17.2 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);
```

The messages are dispatched at the receive side using the `ipc_process_raw_pak()` function.

```
void ipc_process_raw_pak(paktype *pak);
```

Send IPC Messages: Overview

The acknowledged send in IPC, `ipc_send_message_blocked()`, is built on the unacknowledged send, `ipc_send_message()`. The `ipc_send_message_blocked()` function sends an IPC message by using the `ipc_send_message()` function and then initializes a watched boolean variable associated with the IPC message and waits for the boolean to be set to true. The IPC seat manager will set the watched boolean to true and pass a return code when an acknowledgement is received, or a timer times out for the message.

Figure 8-18 Current Tx IPC Message Processing through IPC Seat

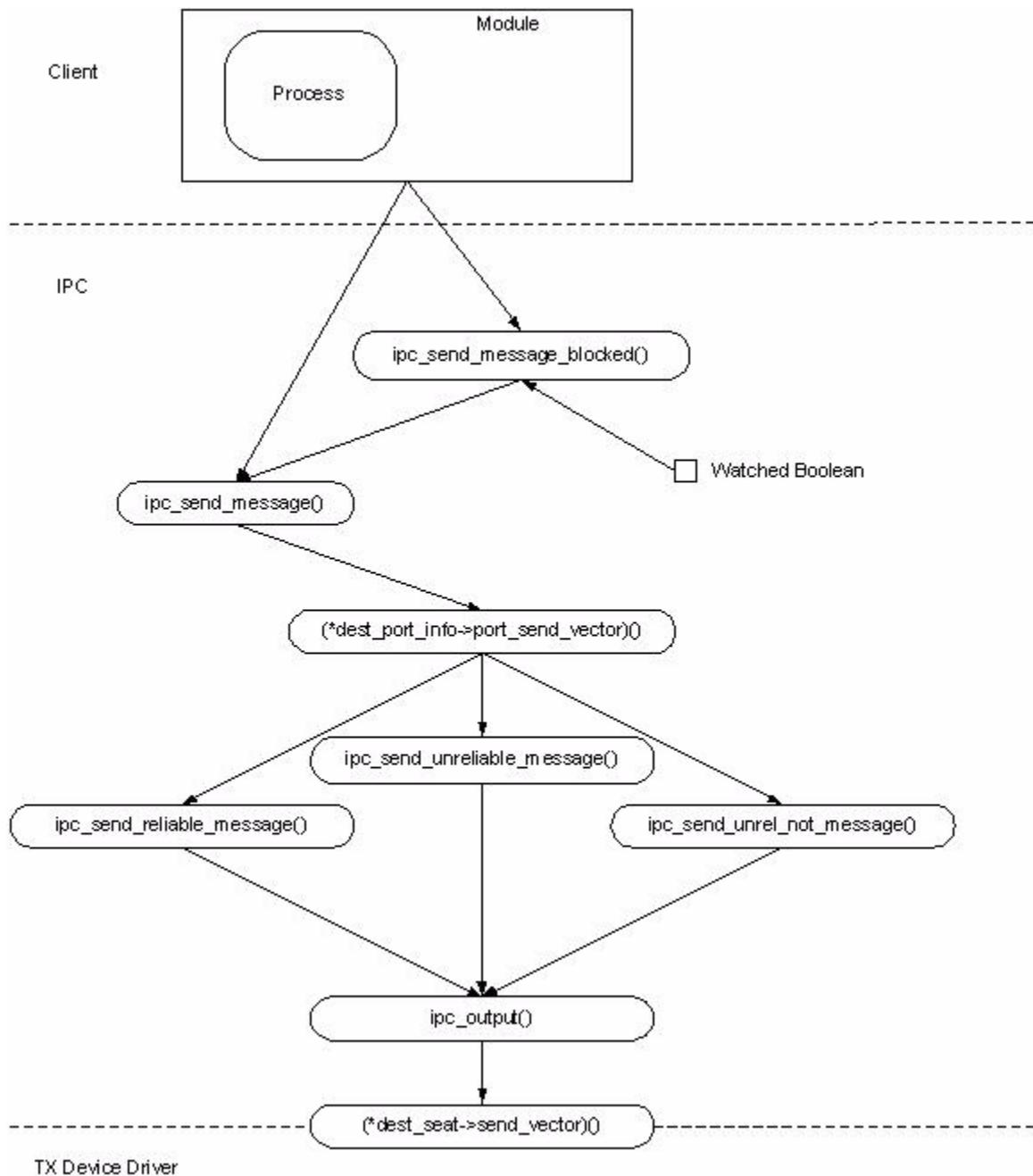


Figure 8-18 shows the current Tx IPC message processing through an IPC seat.

When the unacknowledged send in IPC, `ipc_send_message()` is used, the function passes the IPC message to the appropriate send function for a given IPC session. The send functions for reliable, unreliable, and unreliable with notification transport are `ipc_send_reliable_message()`, `ipc_send_unreliable_message()`, and `ipc_send_unrel_not_message()` respectively. A pointer to the appropriate function is held in a structure used to reference the IPC session, `dest_port_info->port_send_vector`. The IPC session structure is initialized or created while establishing an IPC session.

Each send function sets the appropriate IPC message flags in the IPC message, sets up the necessary IPC timers for the IPC message, queues the message for retransmission if necessary, and passes the IPC message to the device driver by using the `ipc_output()` function.

Device drivers for each of the destination seats are setup during the discovery phase of IPC initialization. Device drivers and destination IPC seats are tightly coupled in IPC.

8.17.3 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);
```

8.18 Receive IPC Messages: Overview

IPC clients start receiving messages after they create their own IPC port. At the port creation, the IPC client specifies the method by which they wish to receive the IPC message. An IPC client may receive an IPC message through a watched message queue (`IPC_QUEUE`), a callback function called from the IPC seat manager (`IPC_CALLBACK`), or a callback function called from the IPC Receive Handler. (`IPC_FAST_CALLBACK`).

If a callback method is used, each callback function has a context data pointer that is passed when the callback function is called. The receive context data pointer is also set at IPC port creation.

8.18.1 Receive a Message

The main entry point for receive processing in IPC is through the `ipc_process_raw_ipc()` function. The exception to this is the UDP IPC interface that has a separate entry point, but that too eventually calls the `ipc_process_raw_ipc()` function. The UDP interface traps the incoming IPC to handle messages destined for the UDP device driver rather than an IPC client. If the `IPC_FLAG_BOOTSTRAP` flag is set in an IPC message, the UDP IPC interface processes it for IPC discovery messages. If the IPC message does not have the `IPC_FLAG_BOOTSTRAP` flag set, the IPC message is passed through the normal IPC message processing functionality. The UDP IPC interface may not work on some platforms because they attempt to use the `IPC_FLAG_BOOTSTRAP` message flag as well.

The main IPC receive entry point `ipc_process_raw_ipc()` function is called either by using a registry call or a direct function call from a device driver. The `ipc_process_raw_ipc()` function is interrupt-safe. The device driver passes the `ipc_process_raw_ipc()`, a PAK message that is decoded into an IPC message for further processing through the `ipc_process_message()` function.

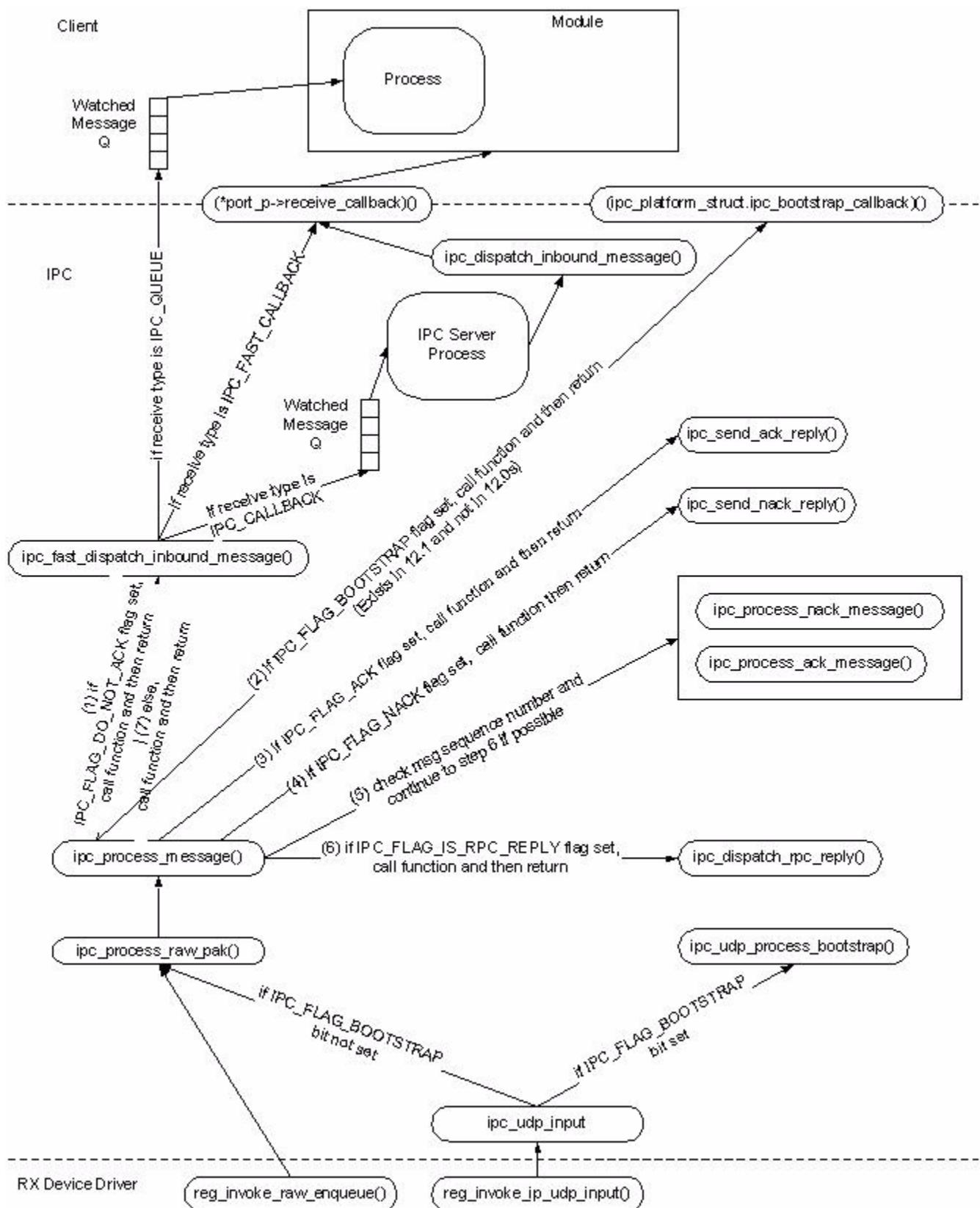
Receive IPC Messages: Overview

The IPC message is processed by using the `ipc_process_message()` function in the following manner:

- 1 If the `IPC_FLAG_DO_NOT_ACK` flag is set in the IPC message header, `ipc_process_message()` function, skip to step 7.
- 2 In IPC Release 12.1, if the `IPC_FLAG_BOOTSTRAP` flag is set in the IPC message header, the IPC message is passed to a platform-dependent IPC message handler and the function exits. If the platform-dependent message handler does not exist, the message is freed and the function exits.
- 3 The `IPC_FLAG_ACK` flag is set in the IPC message header and the IPC message is an acknowledgement for a previously sent reliable IPC message. The destination IPC address indicates the IPC port for which the acknowledgment is meant. If the port exists, the port is updated. The IPC message is freed and the function exits.
- 4 The `IPC_FLAG_NACK` flag is set in the IPC message header and the IPC message is a negative acknowledgement for a previously sent unreliable with notification IPC message. The destination IPC address indicates the IPC port for which the negative acknowledgment is meant. If the port exists, the port is updated. The IPC message is freed and the function exits.
- 5 If the IPC message reaches this step, it is either from a reliable connection, or an unreliable with notification connection. The `ipc_process_message()` function checks the sequence number of the IPC message against the expected sequence number and determines if the message is an old message, an out-of-sequence message, or the expected message.
 - If the `IPC_FLAG_NO_SEQ_ACK` flag is set and the message is an old message, the IPC message is freed and the function exits.
 - If the `IPC_FLAG_NO_SEQ_ACK` flag is set and the message is out-of-sequence, a negative acknowledgement message is sent back to the sender, the IPC message is freed, and the function exits.
 - If the `IPC_FLAG_NO_SEQ_ACK` flag is set and the message is expected, the function continues to the next step.
 - If the `IPC_FLAG_NO_SEQ_ACK` is not set and the message is an old message, and an acknowledgement message is sent to the sender for the received message, the IPC message is freed, and the function exits.
 - If the `IPC_FLAG_NO_SEQ_ACK` is not set and the message is an out-of-sequence message, the IPC message is freed, and the function exits.
 - If the `IPC_FLAG_NO_SEQ_ACK` is not set and the message is expected, the function continues to the next step.
- 6 If the `IPC_FLAG_RPC_REPLY` flag is set, the IPC message is passed to the `ipc_dispatch_rpc_reply()` function and exits the `ipc_process_message()` on return.
- 7 Once all the IPC message processing has been completed, the IPC message is passed to the `ipc_fast_dispatch_inbound_message()` function to be delivered to an IPC client. The `ipc_process_message()` function exits on return.

Figure 8-19 shows the current receive IPC message processing through the seat to the client.

Figure 8-19 Current Receive IPC Message Processing through Seat to Client



The `ipc_fast_dispatch_message()` passes a received IPC message to an IPC client by using

the receive method for the destination IPC port. If the receive method is `IPC_QUEUE`, the IPC message is enqueued onto a watched message queue for the IPC client. If the receive method is `IPC_FAST_CALLBACK`, a callback function to the IPC client is called from the IPC receive handler interrupt context. If the receive method is `IPC_CALLBACK`, a callback function to the IPC client is called from the IPC seat manager process.

8.19 Simulate RPCs

You can use 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,
                                         ipc_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,
                                   ipc_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 10 seconds.

```
void ipc_set_rpc_timeout (ipc_message *message, int seconds);
```

8.20 Congestion Status Notification Capability

You need to register platform-specific applications to enable congestion status notification. Congestion status notification capability (known as the “Backpressure” mechanism) was developed to provide the congestion status of resources from IPC to allow registered clients to handle congestion more effectively.

The mechanism used for congestion status notification is a feedback mechanism that sends a notification from IPC to its registered clients to inform those clients that some resource threshold limits have been reached and indicate that they must stop sending or may resume sending messages. The thresholds are established to try to ensure that IPC clients do not deplete IPC platform resources, such as buffers, causing dropped packets. It is an implementation that polices client send operations, dynamically throttling the send operations based on the available resources and the demand. This mechanism was designed because resource depletion has been observed on both the 7500 and ESR platforms’ IPC implementations because of message traffic between RPs from CEF and Checkpointing on SSO systems.

The `ipc_add_flow_control()` function provides congestion status notification capability to IPC clients, such as Checkpointing or CEF. The API allows the clients to register with IPC so that they will be informed when resource threshold limits have been crossed. Your `ipc_flow_handler` function must know how to stop and resume message send operations when it receives the flow control signals sent via `reg_invoke_ipc_flow_control()` from IPC.

The API exists on all Cisco IOS implementations based on the SSO code base, but behavior is platform-dependent. The API exists on the IOS-based LCs as well as the IOS-based RPs. The clients will register on the LCs, but since the implementation is not present to support the congestion status notification on those platforms, IPC will not call `reg_invoke_ipc_flow_control()`. The effect is that the clients will not see any signals in these environments. The RPs are seen to be the initial focus of this solution (as the problem is not being observed on the line cards), so the congestion status notification support will initially be implemented on the ESR, GSR, and 7500 RPs.

For more information on the IPC RMI feature (for resource management in the latest code) refer the following URL:

<http://wwwin-eng.cisco.com/Eng/IOSTech/WWW/Infrastructure/os-ipc/blogbased/index.html>.

8.20.1 IPC Flow Control Signals

The IPC flow control signals are `IPC_STOP_SENDING` and `IPC_START_SENDING`. These signals are described in the `ipc_add_flow_control()` reference page.

8.20.2 Sending Critical Messages During the STOP State

Critical messages can always be sent after the `IPC_STOP_SENDING` signal has been issued. Currently, only IPC control messages are allowed in IPC, even if the flow-control is hit.

8.20.3 Congestion Status Notification Implementation

To implement the congestion status notification capability, complete the following:

- 1 Include the congestion status notification capability as follows:

```
static watched_boolean *test_ipc_flow_control_boolean = NULL;

static void test_ipc_flow_control (ipc_state state)
{
    boolean on;

    on = (state == IPC_START_SENDING) ? TRUE : FALSE;
    if (test_ipc_flow_control_boolean) {
        process_set_boolean(test_ipc_flow_control_boolean, on);
    }
}

static process test_ipc_traffic_process (void)
{
    boolean our_state;

    <snip>
    test_ipc_flow_control_boolean = create_watched_boolean("IPC Test Flow"
                                                          "Ctrl Bool",
                                                          0);
```

- 2 To subscribe the IPC client for congestion status notification capability, register the application with IPC as follows:

```
/*
 * Register application with IPC to be notified of the backpressure
 * state.
 */
state = ipc_add_flow_control(test_ipc_flow_control,
                            "test_ipc_flow_control");
```

- 3 Establish control of send operations for the application depending on the state of the IPC at the moment. For example:

```
/*
 * Set our state variable appropriately.
 */
our_state = (state == IPC_START_SENDING) ? TRUE : FALSE;

process_set_boolean(test_ipc_flow_control_boolean, our_state);
process_watch_boolean(test_ipc_flow_control_boolean, ENABLE, RECURRING);

while (TRUE) {
    if (process_get_boolean(test_ipc_flow_control_boolean)) {

        /*
         * Send messages here.
         */
    }
}
```

In this example, the boolean depicts the current state of the IPC to the application. The boolean will be modified by the handler which the application registered with IPC. If the current IPC state is `IPC_START_SENDING` (meaning IPC allows clients to send messages), then the application can set the boolean to a `TRUE` value. If the state is `IPC_STOP_SENDING`, then the application can set it to a `FALSE` value.

8.20.4 Policing the IPC Congestion State

The initial implementation of the congestion status notification mechanism is based on the honor system. The clients need to be well behaved for optimum resource usage.

Note The initial implementation uses a list registry. In the first phase, a per-client context is not implemented for the registry. In order to simplify things, an `IPC_STOP_SENDING_WARNING` signal was not implemented in the first phase because if the warning signal was sent, it would be sent to all registered clients on the list registry. However, clients can code to expect this signal in their flow control handler at any time. The enumerated value is included in the initial API header file.

8.20.4.1 Sending the STOP_SENDING_WARNING Signal

In a subsequent implementation, a warning signal will be sent when the client violates the current state. There are no plans to enforce the state, at least for clients of IPC that use the existing IPC APIs. The `IPC_STOP_SENDING_WARNING` signal will be sent via a `reg_invoke_ipc_flow_control` (`IPC_STOP_SENDING_WARNING`) call when a client sends a non-critical message after being told

not to via a previous `reg_invoke_ipc_flow_control (IPC_STOP_SENDING)` call. In the initial implementation, IPC will attempt to send messages even when the client does not honor the IPC congestion status notification.

8.20.5 How are Nonsubscribing IPC Clients Handled?

Though not mandatory, it is highly recommended that all IPC-intensive clients adhere to this congestion status notification framework. However, IPC will continue to allow data send operations from nonsubscribing IPC clients with the available transport resources. If all available resources have been consumed, subsequent send operations, including critical ones, will fail as they do in the current implementation.

The initial clients are Checkpointing and CEF, which are the two chief consumers of IPC resources in an SSO system. The number and size of the resources made available to IPC is being determined experimentally on a platform basis. The implementation was done this way so as to be able to continue to support current IPC clients without the need to modify them while addressing those clients that have been observed to cause failures. Both the resources and the thresholds will be defined to accommodate a reasonable boundary to support nonsubscribing clients and critical messages in the initial phases of implementation.

8.20.6 How Long Can IPC Remain Congested?

The simple answer is: as long as it takes to reclaim sufficient resources to exit the congestion. There is no purpose to using a dead-timer or another means to exit the congestion, because there is no way to create the necessary resources if they are not there. It is assumed that the thresholds will be defined and managed in such a way that critical messages will always be able to be sent.

Some detection mechanism to deal with especially long periods without sufficient IPC resources may need to be implemented. The initial approach will be to reset the Standby under this condition on the assumption that the physical transport mechanism for IPC is somehow broken or blocked. This approach may need to be modified with experience.

8.21 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

8.21.1 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)

```

8.21.2 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;
}

```

8.21.3 Send a Message

You can send a message either in blocking or r mode.

8.21.3.1 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);

```

8.21.3.2 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);
}

```

8.22 Implementing IPCs on the RSP Platform

This section discusses some of the specifics for implementing IPCs on the RSP platform.

8.22.1 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 $2n$, where n is the number of IPC cards that are present in the IPC system. The size of each buffer is 4 KB, 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.

8.22.2 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

8.22.2.1 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.

```
#define IPC_CARD_PRESENT 0x1           /* Set if IPC card is present. */

typedef struct ipc_cbus_card_ {
    uint control;                      /* Control information */
    int slotnum;                       /* Slot number of the IPC card */
    ctrlr_type_t ctrlr_type;           /* Controller type */
    int seat_number;                   /* Assigned seat number */
    rcv_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_ {
    boolean is_cbus_master;           /* Set to TRUE if we are cbus master */
    boolean is_cbus_slave;           /* Set to TRUE if we are cbus slave */
    queueType messageQ;              /* Queue of input messages */

} ipc_cbus_rec_t;

ipc_cbus_rec_t ipc_cbus_rec;
```

8.22.2.2 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.

In the Initialization phase, the init process initializes the IPC sub-system and if available, the IPC UDP sub-system.

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

The entry point for the IPC sub-system is `ipc_subsys_init()`. The outline for the initialization phase is given below. It starts by setting the default values for most fields, that may then be overwritten in the platform-defined `ipc_platform_init()` function. The `ipc_platform_init()` function will define further fields within the IPC global and platform information structures. The databases to store the new IPC ports and IPC seats are not created later. For a platform to perform additional registration after initialization, a function callback, `platform_init_slaves_vector`, and a flag, `do_delay_initialization`, can be set in the IPC platform information structure to have the function called once by the IPC Periodic Timer process.

The outline for initialization phase:

- Allocate IPC global information structure and initialize it to 0.
- Allocate PAKTYPE memory pool.

- Set the IPC Master processes to NO_PROCESS.
- Set the IPC Seat to be the IPC master.
- Create the IPC Seat and IPC Master watched message queues.
- Allocate IPC Port information structures.
- Set the default message memory pool and cache size.
- Call the `ipc_platform_init()` function.
- When the `ipc_platform_init()` function returns, the IPC seat, port, message databases, and memory pools are created.
- The IPC Master IPC ports and processes are created.
- An entry in the IPC Seat database for the local IPC seat is created. This entry is used for IPC messages sent to the local IPC ports.
- Some of the global IPC timers are initialized.
- The registration of IPC callback functions to the protocol stack is done.
- CLI commands are added to the parse graph.
- The IPC Periodic and IPC Seat Manager processes are created. If a control port registration function, or delayed initialization function was registered by `ipc_platform_init()`, those functions will be called by the IPC periodic process.
- Finally, the IPC Master control and echo ports are created.

In the table below, the fields in bold are initialized by IPC prior to calling the `ipc_platform_init()` function. Fields in italics are expected to be set in `ipc_platform_init()`. Fields that are underlined are initialized after `ipc_platform_init()` is called. Fields not set in the `ipc_global_info_` structure have a value of 0 when `ipc_platform_init()` is called. Fields not set in the `ipc_platform_info_` structure have an undefined value when `ipc_platform_init()` is called.

Local IPC information is given below:

```

typedef struct ipc_global_info_ {
    boolean   is_ipc_master;
    watched_queue *inboundQ;
    watched_queue *zone_inboundQ;
    sys_timestamp ipc_periodic;

    ipc_seat   local_seat_addr;
    ipc_seat   last_seat_addr;
    ipc_port   last_port;
    ipc_port_id zone_manager_port;

    int zone_manager_pid;
    int my_zone_manager_pid;
    ipc_port_info *master_info;

    ipc_port_id my_server_port;
    chunk_type *port_info_chunk;
} /* */

/* Good Statistics */
uint      received;

```

/* TRUE if we're a master server */
 /* Raw Input IPC packets */
 /* Raw Input IPC packets */
 /* Next periodic server check */

 /* Our local address */
 /* last address we assigned */
 /* last port we assigned */
 /* address for zone manager PortID */

 /* Where other servers reach me */
 /* Chunk pool for IPC Port structures */

 /* Inbound messages received */

```

        uint      sent;                      /* Outbound messages sent */
        uint      delivered;                 /* Messages delivered to local ports
        uint      acks_in;                  /* Acknowledgements received */
        uint      acks_out;                 /* Acknowledgements sent */
        uint      nacks_in;                 /* NACKs received */
        uint      nacks_out;                /* NACKs sent */

        /* IPC Pak-Type Message Pool */
        int       pool_group;               /* group num of Pool for IPC */
        pooltype *buffer_pool;             /* Buffer Pool for IPC */

        /*
         * Bad Statistics
         */
        uint      input_dropped;            /* IPC frames dropped on input */
        uint      output_dropped;           /* IPC frames dropped on output */
        uint      no_port;                 /* Packets for invalid ports */
        uint      no_seat;                 /* Packets for invalid seat */
        uint      no_transport;             /* Packets for seat w/o transport */
        uint      no_delivery;              /* No queue or No callback installed
        */
        uint      dup_ack;                 /* Duplicate ACK messages received */

        uint      retries;                 /* Retry attempts */
        uint      timeouts;                /* Message timeouts */
        uint      ipc_outputfails;          /* ipc_output returned bad sts */
        uint      message_too_big;          /* IPC message larger than buffer */
        uint      nomessage_buffer;         /* message cache empty */
        uint      no_pak_buffer;             /* couldn't allocate message pak */
        uint      no_memd_buffer;            /* couldn't allocate memd buffer */
        uint      no_emer_message_buffer;    /* Emer. message cache empty */
        uint      no_hwq;                   /* slot doesn't have hwq configed */
        uint      no_orig_in_queue;          /* There as no original message found
                                             for the RPC reply */
        uint      failedOpens;              /* Number of failed opens */
        uint      hardware_error;            /* hardware misbehaved */
        uint      deferred_closure;          /* Counter of deferred port closure
        */

        /*
         * Error simulation
         */
        uint      drop_tx_count;             /* IPC pkt is dropped when this
                                             decrements to 0 */
        uint      cfg_drop_tx_count;          /* user configured drop count */
        boolean   drop_next_tx;              /* TRUE if next IPC pkt must be
                                             dropped */
    } ipc_global_info;
}

```

Platform IPC Seat Information is given below:

```

typedef struct ipc_platform_info_ {
    ipc_transport_type     transport_type;/ * Method used to get there */
    ipc_transport_t        ipc_transport;    /* Seat driver transport info.
*/
    ipc_send_vector        platform_tx_vector; /* transmit to CBUS complex
*/
/* What features are supported by this platform */

```

```

        uint          platform_feat_flags;           /* See below */

        boolean       do_delay_initialize;
        ipc_init_slaves_vector platform_init_slaves_vector; /* init slaves */

        ipc_port_id      master_control_pid;
        ipc_seat         platform_seat;
        char            *platform_seat_name;

        /* Message handler for BOOTSTRAP messages.*/
        ipc_bootstrap_callback_vector_t          ipc_bootstrap_callback;

        /* The number of buffers in the cache */
        int             ipc_message_cache_size;
        /* The max size for IPC Message cache */
        int             ipc_message_cache_max_size;
    } ipc_platform_info;

/*
 * Platform feature flags
 */
#define IPC_PLAT_FEAT_MCAST      0x00000001      /* Multicast able */

```

When the IPC set up is complete, IPC waits for the rest of the initialization to complete before continuing onto the next phase, the discovery phase.

8.22.2.3 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

8.22.3 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.

8.22.4 Microcode Reload Handling

Microcode reloading should be transparent to the IPC core operation.

8.22.5 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.

8.22.5.1 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.

8.22.5.2 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.

8.22.6 IPC Name Service

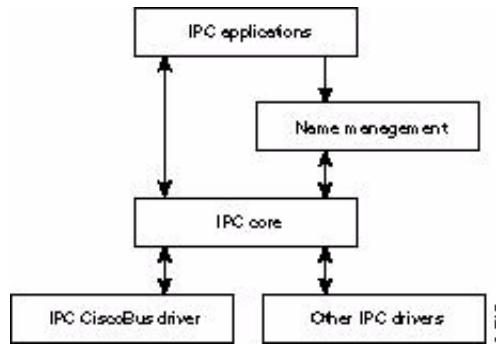
IPC applications are aware of only the names of the remote IPC applications with which they want to communicate. These names need to map to a port identifier so that messages can actually be delivered. This mapping is provided by the *IPC name service*.

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 the 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. Figure 8-20 illustrates the ultimate layered structure of these components.

Figure 8-20 shows the IPC application structure.

Figure 8-20 IPC Application Structure

8.23 Useful Commands for Debugging an IPC Component

The CLI commands that are useful for debugging are **clear ipc stat** and **show ipc status**. The existing **sh ipc status** behavior is changed only if **clear ipc stat** is used to reset the CLI. The **show ipc session tx verbose** command shows the tx side per session details and the **show ipc session rx verbose** command shows the receive side per session details.

8.23.1 clear ipc stat

This command:

- adds the statistic information `ipc_globals` to `ipc_globals_cache`
- resets the `ipc_globals` value
- sets the `ipc_globals` reset time

8.23.2 show ipc status

This command:

- displays IPC statistics from the last reset (i.e. **clear ipc stat**)
- also displays last reset time
- if the **clear ipc stat** command was not issued before, then **sh ipc stat** will display all the IPC statistics since router boottime, which is the same as **sh ipc stat cumulative**

Here is an example of the output from the **show ipc status** command:

```

IPC System Status

Time last IPC stat cleared : never

This processor is the IPC master server.
Do not drop output of IPC frames for test purposes.

1000 IPC Message Headers Cached.
  
```

Total Frames	Rx Side	Tx Side
	189	140

CISCO HIGHLY CONFIDENTIAL

Useful Commands for Debugging an IPC Component

Total from Local Ports	189	70	
Total Protocol Control Frames	70	44	
Total Frames Dropped	0	0	
Service Usage			
Total via Unreliable Connection-Less Service	145	0	
Total via Unreliable Sequenced Connection-Less Svc	0	0	
Total via Reliable Connection-Oriented Service	44	70	
IPC Protocol Version 0			
Total Acknowledgements	70	44	
Total Negative Acknowledgements	0	0	
Device Drivers			
Total via Local Driver	0	0	
Total via Platform Driver	0	70	
Total Frames Dropped by Platform Drivers	0	0	
Reliable Tx Statistics			
Re-Transmission	0	0	
Re-Tx Timeout	0	0	
Rx Errors			
Unsupp IPC Proto Version	0	Tx Session Error	0
Corrupt Frame	0	Tx Seat Error	0
Duplicate Frame	0	Destination Unreachable	0
Out-of-Sequence Frame	0	Tx Test Drop	0
Dest Port does Not Exist	0	Tx Driver Failed	0
Rx IPC Msg Alloc Failed	0	Ctrl Frm Alloc Failed	0
Unable to Deliver Msg	0		
Tx Errors			
IPC Msg Alloc	0	IPC Open Port	0
Emer IPC Msg Alloc	0	No HWQ	0
IPC Frame PakType Alloc	0	Hardware Error	0
IPC Frame MemD Alloc	0		
Buffer Errors			
IPC Msg Alloc	0	IPC Open Port	0
Emer IPC Msg Alloc	0	No HWQ	0
IPC Frame PakType Alloc	0	Hardware Error	0
IPC Frame MemD Alloc	0		
Misc Errors			
IPC Msg Alloc	0	IPC Open Port	0
Emer IPC Msg Alloc	0	No HWQ	0
IPC Frame PakType Alloc	0	Hardware Error	0
IPC Frame MemD Alloc	0		
Tx Driver Errors			
No Transport	0		
MTU Failure	0		
Dest does not Exist	0		

The fields/columns in the **show ipc status** command output are described here:

Time last IPC stat cleared = Displays the time, in dd:hh:mm (or never), since the IPC statistics were last cleared.

This processor is = Shows whether the processor is the IPC master or an IPC slave.

IPC Message Headers Cached = Number of message headers available in the IPC message cache.

Rx Side = Information about IPC messages received.

Tx Side = Information about IPC messages sent.

Service Usage = Number of IPC messages received or sent via connectionless or connection-oriented protocols.

IPC Protocol Version 0 = Number of acknowledgements and negative acknowledgements received or sent by the system.

Device Drivers = Number of IPC messages received or sent using the underlying device drivers.

Reliable Tx Statistics = Number of IPC messages that were retransmitted or that timed out on retransmission using a reliable connection-oriented protocol.

Rx Errors = Number of IPC messages received that displayed various internal frame or delivery errors.

Tx Errors = Number of IPC messages sent that displayed various transmission errors.

Buffer Errors = Number of message allocation failures from the IPC message cache, IPC emergency message cache, IPC frame allocation cache, and IPC frame memory allocation cache.

Misc Errors = Various miscellaneous errors that relate to the IPC open queue, to the hardware queue, or to other hardware failures.

Tx Driver Errors = Number of messages that relate to IPC transmission driver failures including messages to or from a destination without a valid transport entity from the seat; number of messages dropped because the packet size is larger than the maximum transmission unit (MTU); and number of messages without a valid destination address.

8.23.2.1 show ipc status cumulative

The **cumulative** keyword displays the IPC statistics that have been generated since the router was rebooted. Here is an example of the beginning of the output from the **show ipc status cumulative** command:

```
IPC System Status

Time last IPC stat cleared : 00:00:05

This processor is the IPC master server.
...
```

The output has the same fields as for the **show ipc status** command.

8.23.3 show ipc queue

Here is an example of the output from the **show ipc queue** command:

```
There are 28 messages currently reserved for reply msg.
Source          Destination          Size   Tries   Seq
10000.0        IPC Master         10000.5    Card5/0:Request 968   -1   37065
10000.0        IPC Master         10000.5    Card5/0:Request 44    -1   37066
...
```

The first two columns (1, 2) indicate the *seat.port* number and the *name* of the source port (IPC Master in this example).

The second two columns (3, 4) indicate the *seat.port* number and the *name* of the destination port (Card5/0:Request in this example).

The 5th column indicates the *size* of the message.

The 6th column indicates the *retry* number (-1 means the message is not eligible for transmission; the *retry* number is set to -1 when using fragmentation).

The last column indicates the *sequence* number.

8.23.4 show ipc ports

This command provides port-level statistics.

Here is an example of the output from the **show ipc port** command:

```
There are 59 ports defined.
 10000.1      unicast          IPC Master:Zone
 10000.2      unicast          IPC Master:Echo
 10000.3      unicast          IPC Master:Control
 10000.4      unicast          Rf_Proxy:Server:5
port_index = 0  seat_id = 0x2150000  last sent = 0      last heard = 175
0/2/175

 10000.5      unicast          Card5/0:Request
port_index = 0  seat_id = 0x10000  last sent = 0      last heard = 37064
0/827/48468168
```

The first column is the *seat.port* number of the port.

The second column is the *type* (unicast in this example).

The third column is the *name*.

The lines that begin with “*port_index*” show the session information for the previous port. For example, in the session information for the port 10000.5, 0/827/48468168 shows *rcv_pending/rcv_pending_peak/rcv_total*.

8.23.4.1 show ipc session

This command provides session-level statistics: **show ipc session tx|rx|all verbose**

You must specify **tx**, **rx**, or **all** to see only the transmit-side, only the receive-side, or all sessions, respectively.

8.23.5 show ipc nodes

Here is an example of the output from the **show ipc nodes** command:

```
There are 6 nodes in this IPC realm.
```

ID	Type	Name	Last Sent	Last Heard
0.10000	Local	IPC Master	0	0
0.1060000	RSP-CY	RSP IPC card slot 6	9	79
0.1050000	RSP-CY	RSP IPC card slot 5	21	22

0.1080000	RSP-CY	RSP IPC card slot 8	21	22
1.10000	Local	IPC Master: -Zone#1	0	0
2.10000	Local	IPC Master: -Zone#2		

The first column is the *zone.seat* ID for the seat.

The second column is the *type*.

The third column is the *name*.

The fourth column is the *sequence* number of the message that was last sent.

The fifth column is the *sequence* number of the in-sequence message that was last heard.

8.23.6 show ipc rpc

This command shows how many RPC requests and replies have been sent and received by the default zone. Also, in the output you can see counters for RPC-specific errors and the last time that these counters were cleared.

Here is an example of the output from the **show ipc rpc** command:

```
v1-7500-1#show ipc rpc
          IPC RPC Status
          Rx Side      Tx Side
Total RPC requests           1426        72
Total RPC replies            72         1426

          IPC RPC Errors
No Pending RPC Request for Received RPC Reply      0
RPC Request which have timeout out                0

Time last IPC RPC stat cleared                  00:00:00
```

8.23.7 show ipc zones

Here is an example of the output from the **show ipc zones** command:

There are 3 Zones in this IPC realm.

Zone ID	Seat ID	Name
0	10000	IPC Default Zone
1	10000	IPC TEST ZONE#1
2	10000	IPC TEST ZONE#2

The first column is the *number* for the zone.

The second column is the *seat* for the zone.

The third column is the *name* for the zone.

8.23.8 show ipc session tx verbose

Here is an example of the output from the **show ipc session tx verbose** command:

Tx Sessions:		
Port ID	Type	Name
10000.5	Unicast	ERP Agent Port #5

CISCO HIGHLY CONFIDENTIAL

Useful Commands for Debugging an IPC Component

```
port_index = 0 type = Reliable           last sent = 11      last heard = 0
session creator pc = 4201979C ipc_tx_context = 0x0
Msgs requested      = 11                  Msgs returned      = 11
receiver seq-window size      = 128

Basic Statistics:
    Total Tx Frames          = 11      Total Rx Frames
= 11
        Total Tx Frames Sent      = 11      Total Rx Frames
Delivered      = 0
        Total Tx via Platform Drivers      = 11      Total Rx via Platform
Drivers      = -
        Total Tx Acknowledgements      = 0      Total Rx
Acknowledgements      = 11
        Total Tx Frames Dropped      = 0      Total Rx Frames Dropped
= 0
        Last Tx Timestamp      = 00:22:03 Last Rx Timestamp
= 00:22:03

Tx Errors:
    Total Frame Driver Failures      = 0      Total Quiesced
= 0
    Total Test Drops      = 0      Total Session Errors
= 0
    Total Seat Errors      = 0      Total Send Errors
= 0
    Total Re-Tx Timeout      = 0      Total Re-Transmission
= 0

Memory Statistics:
    Total Tx Memory Alloc Failure      = 0
    No Tx Message Buffer      = 0      No Tx Emergency Message
Buffer      = 0
    Message Too Big      = 0      Largest Message Size
= 36

RPC Statistics:
    Total RPC Requests Out      = 11      Total RPC Requests In
= 0
    Total RPC Replies Out      = 0      Total RPC Replies In
= 11
    No RPC Request for Rx RPC Reply = 0      RPC Timeouts
= 0

Message Bundling Statistics:
    Total Tx Bundled Messages      = 0      Total Tx Bundled
SubMessages      = 0
    Total Tx NonBundled Messages      = 11      Total Tx Urgent Messages
= 11
    Total Tx Bundled Errors      = 0

Flow Control Statistics:
    Session ACK pending      = 0      Session ACK pending peak
= 1
    Seat ACK pending      = 0      Seat ACK pending peak
= 20
    Total Sess level stop sent      = 0      Total Seat level stop
sent      = 0
```

```

        Total Seat level crit rising/crit falling/hi rising/hi falling =
0/0/0/0
        Total Peer stop sent          = 0
        Total no. of non-conformant msgs = 0      Max size of
non-conformant msgQ = 0
        Number of queued messages     = 0      Current flow control
state      = 0x0
        Current seat flow ctrl state = 0x0      Current sess flow ctrl
state      = 0x0
        Current ctrl ack not flowchecked = 2

```

8.23.9 show ipc session rx verbose

Here is an example of the output from the **show ipc session rx verbose** command:

```

Rx Sessions:
Port ID      Type      Name
2150000.5    Unicast   Rf_Proxy:Client:5

port_index = 0  seat_id = 0x10000    last sent = 0      last heard = 1274
No of msgs requested      = 1274  Msgs returned       = 1274
seq-window size           = 128   number of messages in window = 0
maximum out-of-seq received = 0    maximum pending in window = 0
number of messages dropped = 0    number of window resets = 0

Basic Statistics:
Total Tx Frames          = 1274      Total Rx Frames
= 1274
Total Tx Frames Sent      = 0        Total Rx Frames
Delivered      = 1274
Total Tx via Platform Drivers = 1274      Total Rx via Platform
Drivers      = -
Total Tx Acknowledgements = 1274      Total Rx
Acknowledgements      = 0
Total Tx Frames Dropped   = 0        Total Rx Frames Dropped
= 0
Last Tx Timestamp         = 00:22:38 Last Rx Timestamp
= 00:22:38

Errors:
Total Tx Frame Driver Failures = 0      Total Tx Quiesced
= 0
Total Tx Test Drops         = 0        Total Tx Session Errors
= 0
Total Tx Seat Errors        = 0        Total Tx Send Errors
= 0
Total Rx Duplicate Frames   = 0        Total Rx Out of Sequence
Frames      = 0

Memory Statistics:
Total Tx Memory Alloc Failure = 0

RPC Statistics:
Total RPC Requests Out      = 0        Total RPC Requests In
= 1274
Total RPC Replies Out        = 1274     Total RPC Replies In
= 0

```

```
No RPC Request for Rx RPC Reply = 0          RPC Timeouts  
= 0  
  
Message Bundling Statistics:  
    Total Rx Bundled Messages      = 0          Total Rx Bundled  
SubMessages = 0  
    Total Rx NonBundled Messages   = 1274  
    Total Rx Bundled Errors       = 0
```

8.24 IPC Master

The following section describes the IPC Master feature for the IOS IPC protocol. For a complete description of the IOS Master, please see EDCS-245959 and EDCS-264998.

8.24.1 Terms

ACK

Acknowledgement. In IPC, a reliable communication acknowledgement is sent for every packet received.

IPC

Inter-Processor Communication protocol

IPC Client

A software module that uses IPC services

IP Connection

An IPC session is a simplex communication channel between two IPC ports

IPC Port A

Communication end-point within IPC, used as the source and destination of all communication

IPC Seat

An IPC seat is a computational element, such as a processor, that can be communicated using IPC. An IPC seat is where IPC clients and ports reside.

LC

Line Card

NACK

This is used to inform the sender that the packet received is out of sequence

RP

Route Processor

8.24.2 What is the IPC Master?

Presently, IPC can support only one seat per active RP, slave RP and LCs. The IPC server can function either as a master or as a slave. Only one IPC master is supported in this distributed system. All the IPC servers in the system are considered as one group with only one master. IPC cannot support more than one IPC master in a system, so all the communication between RP to Slave RP and RP to LC go through the IPC master. This restricts communication between LC to LC.

Moreover, LCs cannot function as masters to other LCs and as a slave to the RP. This is because the current IPC can only function either as a master or as a slave and only one group of seats can exist in the system.

In order to have direct communication between LCs and RP, we need the support of more than one independent seat group to exist and the support for more IPC masters in the system.

IPC Master allows more than one group of IPC seats to exist in a distributed system, more than one seat in the RP, Slave RP and LCs. Furthermore, more than one IPC seat master can exist in the system. IPC Master facilitates IPC to serve as a master to one group of seats and also serve as a slave to other groups of seats.

8.24.3 IPC Master API Functions

The following sections are discussed in this document:

- Creating a Zone
- Getting a Zone by Name
- Getting Zone Data
- Getting the Zone Name
- Closing All Seats for a Zone
- Removing a Zone
- Setting a Zone's Platform ACK Vector
- Setting a Zone's Platform Master Control Port ID
- Setting a Zone's Platform Seat ID
- Setting a Zone's Platform Seat Name
- Setting a Zone's Platform Transport
- Setting a Zone's Platform Transport Type
- Setting a Zone's Platform TX Vector
- Setting the Seat Master
- Setting a Zone's Local Seat Address

8.24.3.1 Creating a Zone

To create a new ipc zone, call the `ipc_create_zone()` function.

```
#include "ipc_zones.h"
ipc_error_code ipc_create_zone(uint zone_val,
                                ipc_zone *zone_id,
                                char *zone_name)
```

8.24.3.2 Getting a Zone by Name

To get a zone by name, call the `ipc_get_zone_by_name()` function.

```
#include "ipc_zones.h"
ipc_zone_data *ipc_get_zone_by_name(char *name);
```

8.24.3.3 Getting Zone Data

To get zone data, call the `ipc_get_zone_data()` function.

```
#include "ipc_zones.h"
ipc_zone_data *ipc_get_zone_data(ipc_zone zone_addr);
```

8.24.3.4 Getting the Zone Name

To get the zone name, call the `ipc_get_zone_name()` function.

```
#include "ipc_zones.h"
char *ipc_get_zone_name(ipc_zone zone_id)
```

8.24.3.5 Closing All Seats for a Zone

To close all the seats for a zone, call the `ipc_remove_seat_on_zone()` function.

```
#include "ipc_seats.h"
void ipc_remove_seat_on_zone(ipc_zone zone_id);
```

8.24.3.6 Removing a Zone

To remove a zone, call the `ipc_remove_zone()` function.

```
#include "ipc_zones.h"
ipc_error_code ipc_remove_zone(ipc_zone zone_id);
```

8.24.3.7 Setting a Zone's Platform ACK Vector

To set a zone's platform acknowledgement vector, call the `ipc_set_platform_ack_vector()` function.

```
#include "ipc_zones.h"
void ipc_set_platform_ack_vector(ipc_platform_ack_vector platform_ack_vector,
                                 ipc_zone zone_id);
```

8.24.3.8 Setting a Zone's Platform Master Control Port ID

To set a zone's platform master control port ID, call the `ipc_set_platform_master_control_portid()` function.

```
#include "ipc_zones.h"
void ipc_set_platform_master_control_portid(ipc_port_id master_control_pid,
                                            ipc_zone zone_id);
```

8.24.3.9 Setting a Zone's Platform Seat ID

To set a zone's platform seat id, call the `ipc_set_platform_seat_id()` function.

```
#include "ipc_seats.h"
void ipc_set_platform_seat_id(ipc_seat platform_seat,
                             ipc_zone zone_id);
```

8.24.3.10 Setting a Zone's Platform Seat Name

Set a zone's platform seat name, call the `ipc_set_platform_seat_name()` function.

```
#include "ipc_zones.h"
void ipc_set_platform_seat_name(char *platform_seat_name,
                                ipc_zone zone_id);
```

8.24.3.11 Setting a Zone's Platform Transport

To set a zone's platform transport, call the `ipc_set_platform_transport()` function.

```
#include "ipc_zones.h"
void ipc_set_platform_transport(ipc_transport t, ipc_zone zone_id);
```

8.24.3.12 Setting a Zone's Platform Transport Type

To set a zone's platform transport type, call the `ipc_set_platform_transport_type()` function.

```
#include "ipc_zones.h"
void ipc_set_platform_transport_type(ipc_transport_type, ipc_zone zone_id);
```

8.24.3.13 Setting a Zone's Platform TX Vector

To set a zone's platform tx vector, call the `ipc_set_platform_tx_vector()` function.

```
#include "ipc_zones.h"
void ipc_set_platform_tx_vector(ipc_send_vector platform_tx_vector,
                                ipc_zone zone_id);
```

8.24.3.14 Setting the Seat Master

To set the seat master, call the `ipc_set_seat_master()` function.

```
#include "ipc_seats.h"
void ipc_set_seat_master(ipc_seat seat_id, boolean true);
```

8.24.3.15 Setting a Zone's Local Seat Address

To set a zone's local seat address, call the `ipc_set_zone_local_seat_addr()` function.

```
#include "ipc_seats.h"
void ipc_set_zone_local_seat_addr(ipc_seat local_seat_addr,
                                  ipc_zone zone_id);
```

8.25 IPC Get Port Notifications

IPC port notification APIs are used to inform IPC clients of port-level events. Examples of events include close, open, reregister, and reopen.

There are two port notification APIs: `ipc_set_tx_port_event_notify()` and `ipc_set_rx_port_event_notify()`. These APIs are available in the 12.2S, 12.2SRB, 12.2SX, 12.4, and 12.4T. Previously, these APIs were called `ipc_set_port_op_notification()` and `ipc_set_port_op_callback()` respectively.

8.25.1 Getting Notification of a Receiver Port Event

To set a callback function that should be invoked when a port encounters a port event on the receiver side because of OIR or SSO of the port, call the `ipc_set_rx_port_event_notify()` function.

```
#include "ipc_ports.h"
void ipc_set_rx_port_event_notify (ipc_port_id portid,
                                    void *context,
                                    ipc_rx_port_event_notify_handler_t func);
```

8.25.2 Getting Notification of a Sender Port Event

To set a callback function that should be invoked when a session encounters a port event on the sender side because of OIR or SSO of the port, call the `ipc_set_tx_port_event_notify()` function.

```
#include "ipc_ports.h"
void ipc_set_tx_port_event_notify (ipc_port_info *port_info,
                                    void *context,
                                    ipc_tx_port_event_notify_handler_t
                                    handler);
```

8.26 References

For more information on IPC, refer the following links:

IPC homepage:

<http://wwwin-eng.cisco.com/Eng/IOSTech/WWW/Infrastructure/os-ipc/blogbased/index.html>

IPC FAQ: http://wiki-eng.cisco.com/engwiki/IPC_2d_2dFAQandFactoids.

IPC PRRQ: <http://wwwin-tools.cisco.com/prrq/viewQueue.do?queueName=IPC>

IPC FAQ: http://wiki-eng.cisco.com/engwiki/IPC_2d_2dFAQandFactoids

Multi-OS IPC FAQ: http://wiki-eng.cisco.com/engwiki/IPC_2ddev_2fMulti_2dOS

File System

Replaced the NVRAM Multiple File System section with the section 9.7 “Persistent Variable Method”.(July 2010)

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).

Note Cisco IOS file system development questions can be directed to the interest-ifs@cisco.com, codereviewer-ifs@cisco.com, and mwiedman-dev-group@cisco.com aliases.

This chapter includes information on the following topics:

- Accessing File Systems
- Implementing Simple File Systems
- Implementing Complete File Systems
- Additional File System Hooks
- NVRAM API
- Persistent Variable Method
- References

9.1.1 Application Level API

The application level API for `ifs` 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 Systems

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)
fsid_t sifs_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_IWUSR, 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);
struct sifs_vector_ {
    ifs_vector_read_t    read;
    ifs_vector_write_t   write;
    sifs_vector_open_t   open;
    sifs_vector_close_t  close;
};
```

It's 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,
              ACCT_IFS_FEATURE, &acct_ready_root, 1,
              ACCT_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)
boolean sifs_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 by `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
 */
#define ACCT_IFS_FLAGS      (IFS_FLAGS_ACCESS_RO | IFS_FLAGS_PATH_FILENAME)
#define ACCT_IFS_FEATURE     IFS_FEATURE_NONE

#define ACCT_IFS_FILE_NAME   "acctng_file1"
#define ACCT_FILE_INDEX      1 /* 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;

    acct_ready_fsid = sifs_create_fs("atm-acct-ready", IFS_TYPE_OPAQUE,
                                    ACCT_IFS_FLAGS, ACCT_IFS_FEATURE,
                                    &acct_ready_root, 1,
                                    ACCT_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
#define SYSTEM_IFS_BLOCK_SIZE SIXTEEN_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 (subsysstype *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_sysdент *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_sysdент));
    if (!sysdent) {
        errno = ENOMEM;
        return (IFS_FD_ILLEGAL);
    }
    sdent = fdent->fd_context;
    sdent->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;
    uint          i, 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);
}

```

9.3.1.6 Example 4 - Accessing the File System

This example illustrates the behavior of the following three functions:

- `int ifs_iopen(const char *prefix, ino_t ino, int oflags, mode_t mode);`
`ifs_iopen()` is the IFS implementation of a function to open a file, which should have been created prior to calling this function using `ifs_open()`, along with the `O_CREAT` flag, in order to read/write/append/truncate using the inode number/file number.
- `int ifs_istat(const char *path, ino_t ino, struct stat *stat_buf);`
`ifs_istat()` is the IFS implementation for gathering the statistics of a file given its inode number.
- `int ifs_fioctl(int fd, int function, void *arg);`
`ifs_fioctl()` is the IFS implementation for performing an IOCTL function on an open file, which should have been opened prior to calling this function using either `ifs_open()` or `ifs_iopen()` given its file descriptor (fd).

A file descriptor is a small unsigned integer that the IOS Filesystem (IFS) uses to identify a file. A file descriptor is created by a process through issuing an `ifs_open()` or `ifs_iopen()` call for the file name or file number/inode number. A file descriptor ceases to exist after a call to `ifs_close()`.

Here is a code example from `sys/dev/sr_flashmib.c`. It is a modified version of the function `get_exact_ciscoFlashFileEntry()`:

```

static boolean get_exact_ciscoFlashFileEntry (int *devNum, int *fileNum,
fioctl_arg_t *snmp_info, stat *stat)

    boolean success = FALSE;
    /*
     * Name of the filesystem, termed prefix in IOS
     */
    char *devnm;

```

```

int file_cnt;
int fd;

/* beginning of preliminary checks */

devnm = malloc(FLASH_PHY_DEV_NAME_LEN + 1);
if (!devnm)
    return (FALSE);

/*
 * Given the device number and partition numbers, this function makes
 * the character string that refers to the device under consideration.
 * This device name can then be passed to lower level functions to
 * perform operations on or extract information about the device.
 */
snmp_make_devnm(*devNum, 0, devnm);
if (!strlen(devnm)) {
    /* oops, could not make a good name , got to get out. */
    free(devnm);
    return (FALSE);
}

/* Determine number of files on that device */
file_cnt = snmp_flash_partition_filecount(devnm);
if (*fileNum > file_cnt) {
    /*
     * cannot get file info for a 'file' number greater than the maximum
     * number of files on this partition.
     */
    free(devnm);
    return (FALSE);
}
/* end of preliminary checks */
/* do the real business now */
if (ifs_istat(devnm, *fileNum, stat) >= 0) {
/*
 * Using the file number, otherwise known as the inode number, stat that
 * file.
*/
    fd = ifs_iopen(devnm, *fileNum, O_RDONLY, 0);
    /*
     * open the file using the inode number
     */
    if (fd != IFS_FD_ILLEGAL) {
        if (ifs_fioctl(fd, IFS_IOCTL_GET_SNMP_FILE, snmp_info) >= 0){
        /*
         * Perform a IOCTL operation with an open file (using the
         * file descriptor - FD)
         */
            success = TRUE;
        }
        ifs_close(fd);
    }
}
/* free the malloc'd dev name before leaving, we want no memory leaks */
free(devnm);
return (success);

```

9.3.1.7 Example 5 - Reading a File

This example illustrates the behavior of the following function:

```
int ifs_read(int fd, char *buf, size_t nbytes);
```

To read data from a file descriptor, call the `ifs_read()` function.

Here is a sample piece of code from the `ifs_verify_md5()` function in `sys/ifs/ifs_command_verify.c`:

Open up the file using the `ifs_open()/ifs_iopen()` API function and then use this file descriptor for reading from the file, passing the buffer to read into, and reading the size.

```
do {
    read_size = ifs_read(fd, chunk_of_data, READ_SIZE);
    if (read_size <= 0) {
        if (read_size < 0) { /* Error while reading */
            printf(ifs_string_read_error, pathent->path,
                   ifs_strerror(errno));
            ifs_close(fd);
            ifs_pathent_destroy(pathent);
            free(chunk_of_data);
            return;
        }
        break; /* reached end of file */
    }
}
```

9.3.1.8 Example 6 - Writing to a File

This example illustrates the behavior of the following function:

```
int ifs_write(int fd, char *buf, size_t nbytes);
```

To write data to a file descriptor, call the `ifs_write()` function. In CLASS A and CLASS B file systems, if a file is opened for write, it is truncated and the write happens from the start of the file.

Here is a sample piece of code from the `write_remote_rcsf()` function in `sys/filesys/remote_config_sync.c`.

Open up the file using the `ifs_open()/ifs_iopen()` API function and then use this file descriptor for writing into the file, passing in the buffer to write, and writing the size. Compare the return value with the actual size to see if the write was successful in writing all the contents of the passed buffer.

Note Only the CLASS C file system supports *appending* to a file; in all other file systems you can only write from the *start* of the file.

```
static boolean write_remote_rcsf (char *buf, int size)
{
    int retcode;

    if (validate_remote_rcsf() == TRUE) {
        retcode = ifs_write(remote_rcsf_fd, buf, size);
        if (retcode != size) {
            if (retcode < 0) {
                printf("\n%% Secondary Running config write error (%d)\n",
                       retcode);
            } else {
                printf("\n%% Secondary Running config write incomplete
                       (%d)\n", retcode);
            }
            close_remote_rcsf();
            return FALSE;
        }
    }
    return TRUE;
}
```

9.3.1.9 Example 7 - Seeking a Position within a File

This example illustrates the behavior of the following function:

```
int ifs_lseek(int fd, off_t offset, int whence);
```

To seek to a position within a file, call the `ifs_lseek()` function.

Here is a code snippet from the `ifs_download()` function in
`sys/src-m860-les/ipm_dsprm.c`.

Open the file using the `ifs_open()/ifs_iopen()` API, and get the fd. You can then use `SEEK_SET`, `SEEK_END`, or `SEEK_CUR` and specify the offset to move the file pointer from where the next read or write will happen.

```
static uchar* ifs_download (uint8 *fname)
{
    int                 fd;
    uint                byte_cnt, sz;
    uchar              *hdr, *cp;

    if ((fd = ifs_open(fname, O_RDONLY, 0)) < 0) {
        buginf("\n%s: Failed in opening %s for download.",
               __FUNCTION__, fname);
        return (NULL);
    }
    if (!(sz = ifs_fd_get_size(fd))) {
        buginf("\n%s: Empty file %s.", __FUNCTION__, fname);
        ifs_close(fd);
        return (NULL);
    }
    if (!(hdr = cp = malloc_named(sz * sizeof(uchar), "DSPware flash
download")))
    {
        ifs_close(fd);
        return (NULL);
    }
    ifs_lseek(fd, 0, SEEK_SET); /* Skip ELF hdr if any to the DSP image */
    while ((byte_cnt = ifs_read(fd, cp, HPI_SIZE)) == HPI_SIZE) {
        cp += HPI_SIZE;
    }
    ifs_close(fd);
    return (hdr);
}
```

9.3.1.10 Example 8 - Renaming a File

This example illustrates the behavior of the following function:

```
int ifs_rename(const char *frompath, const char *topath);
```

To rename a file, call the `ifs_rename()` function.

Note The `ifs_rename()` function is available only for CLASS C file systems.

Here is a self-documenting code example from the `ifs_rename_file()` function in `sys/ifs/ifs_command_rename.c`.

```
/*
 * ifs_rename_file
 *
 * Renames the file given by the source filename to the destination
 * filename supplied. If successful, it returns the total number of bytes
 * copied. If it fails, it returns zero.
 */

static int ifs_rename_file (char *src_file, char *dst_file)
{
    /*
     * Attempt to rename the file
     */
    if (ifs_rename (src_file, dst_file)) {
        printf(ifs_string_rename_error, src_file, dst_file,
               ifs_strerror(errno));
        return (0);
    }

    return (TRUE);
}
```

9.3.1.11 Example 9 - Removing a File

This example illustrates the behavior of the following three functions:

- `int ifs_remove(const char *path);`
- `int ifs_stat(const char *path, struct stat *stat_buf);`
- `int ifs_fstat(int fd, struct stat *stat_buf);`

To return information about an open file descriptor, call the `ifs_fstat()` function. This function is the same as `ifs_stat()`, except that you need to pass a file descriptor instead of the path/filename.

Here is a code snippet from the `snmp_file_delete()` function in `sys/dev/sr_flashmib.c`.

```
static int snmp_file_delete (ifs_pathent          *pathent,
                           ciscoFlashMiscOpEntry_t *entry)
{
    struct stat      stat_buf;

    /*
     * Find out if the path is a valid one
     */

    if (ifs_stat(pathent->path, &stat_buf) < 0)
    /*
     * There is no need to stat() before you call ifs_remove, to check if the
     * file is present as the ifs_remove code itself does it. Bad code or
     * time to market, eh ?
    */
    /*
     * Use the path and pass the stat_buf to collect the file statistics.
     * Also used for checking whether a file is present.
    */
    return (SNMP_FILE_OPEN_ERR);
```

```
    if (ifs_remove(pathent->path) < 0) <<< Just pass the complete path along
with the filesystem prefix and it does the rest for you
        return (ifs_errno_to_snmp_status());
        return (SNMP_NO_ERROR);
    }

/*
 * The data structure used by stat() and fstat() calls
 */
struct stat {
    /* POSIX fields */
    mode_t      st_mode;
    ino_t       st_ino;
    fsid_t      st_dev;
    nlink_t     st_nlink;
    uid_t       st_uid;
    gid_t       st_gid;
    off_t       st_size;
    time_t      st_atime;
    time_t      st_ctime;
    time_t      st_mtime;
};


```

This data structure is defined in `cisco.comp/ansi/include/sys/stat.h`.

9.3.1.12 Example 10 - Making and Removing a Directory

This example illustrates the following two functions:

```
int ifs_mkdir(const char *path, mode_t mode);
int ifs_rmdir(const char *path);
```

Note Directory support is present only in CLASS C filesystems. *mode* is not currently implemented and, by default, `S_IFDIR` is used.

Pass the full path to create/remove a directory. Only empty directories can be removed.

Here is a code snippet from the case `TAR_TYPE_DIR_TYPE` in the `ifs_archive_tar()` function in `sys/ifs/ifs_command_tar.c`.

```
<snip>
    if (extract_dir) {
        snprintf(buffer, sizeof(buffer), "%s/%s",
                 extract_dir, th.header_info.name);
        if (ifs_mkdir(buffer, 0)) {
            if (errno != EEXIST) {
                if (http_flag) {
                    ifs_file_buginf(" -- unable to create (%s)",
                                    ifs_strerror(errno));
                } else {
                    printf(" -- unable to create (%s)",
                           ifs_strerror(errno));
                }
            }
        }
    }
<snip>
```

9.3.1.13 Example 11 - Collect File System Statistics

This example illustrates the behavior of the following function:

```
int ifs_statfs(const char *path, struct statfs *statfs_buf);
```

This function collects the statistics of the file system. The following code example comes from the `snmp_flash_filecnt()` function in `sys/dev/snmp_flash_cmnds.c`.

Do a `statfs` by passing the file system prefix and a `statfs` buffer (the `struct` definition is given below).

```
/*
 * see how many beasts the puppy is carrying.
 */
unsigned int snmp_flash_filecnt (void)
{
    char *system_flash;
    struct statfs statfs_buf;
    int rc;

    system_flash = reg_invoke_flash_device_default();
    if (!system_flash)
        return (0);
    rc = ifs_statfs(system_flash, &statfs_buf);
    return ((rc < 0) ? 0 : statfs_buf.f_fused + statfs_buf.f_fdel);
}

/*
 * File System status structure
 *
 * CISCO PRIVATE - Not defined by POSIX
 */
```

```

struct statfs {
    long      f_type;          /* File system type */
    long      f_bsize;         /* Size on one block on fs */
    long      f_blocks;        /* Total number of blocks on file system */
    long      f_bfree;         /* Number of free blocks */
    long      f_files;         /* Total number of files possible */
    long      f_ffree;         /* Number of free files slots */
    long      f_fused;         /* Number of good files */
    long      f_fdel;          /* Number of deleted files */
    fsid_t   f_fsid;
    long      f_spare[7];
};

}

```

This structure is defined in `cisco.comp/ansi/include/sys/statfs.h`.

9.3.1.14 Example 12 - IOCTL Function Support and Getting Next Directory Entry

This example illustrates the behavior of the following two functions:

- `int ifs_ioctl(const char *path, int function, void *arg);`

This function performs an IOCTL-like function on a given filesystem. The various IOCTL functions supported are defined in `sys/ifs/ifs.h`.

- `int ifs_getdents(int fd, struct dirent *buf, size_t nbytes);`

This function is used to get the next entry in the directory which has been opened as a file descriptor using the `ifs_open()/ifs_iopen()` API function.

This code example is from the `snmp_file_undelete()` function in `sys/dev/sr_flashmib.c`.

```

int ifs_close(int fd);
/*
 * Close the FD passed to it.
 */

static int snmp_file_undelete (ifs_pathent           *pathent,
                               ciscoFlashMiscOpEntry_t *entry)
{
vint                  fd, index;
struct dirent         dirent;
ifs_ioctl_undelete   undelete;
char                 prefix[IFS_MAX_PREFIX_LENGTH + 2];
strcpy(prefix, pathent->prefix);
strcat(prefix, ":");

if ((fd = ifs_open(prefix, O_RDONLY, 0)) < 0)
    return (SNMP_FILE_OPEN_ERR);

for (index = 1; ; index++) {
/*
 * Find the darn thing....
 */
if (ifs_getdents(fd, &dirent, sizeof(struct dirent)) <= 0) {
/*
 * Loop thru the directory, getting the next directory entry in the
 * dirent struct
*/
    ifs_close(fd);
    return (SNMP_INVALID_SRC_FILE);
}

```

```

        if (strcmp(pathent->filename, dirent.d_name) != 0)
            continue;

        /*
         * Initialize the ioctl block
         */
        undelete.flags = IFS_UNDELETE_FLAGS_AUTO;
        undelete.ino = index;

        /*
         * Now restore the file.
         */
        if (ifs_ioctl(prefix, IFS_IOCTL_UNDELETE, &undelete) < 0) {
        /*
         * Call the ifs_ioctl() function with the IFS_IOCTL_UNDELETE request,
         * a negative value is returned for failures
         */
        ifs_close(fd);
        return (ifs_errno_to_snmp_status());
    }
    break;
}
/* Close the opened file */
ifs_close(fd);
return (SNMP_NO_ERROR);
}

```

9.3.1.15 Example 13 - Closing All Files

This example illustrates the behavior of the following function:

```
void ifs_closeall(int fd_table[]);
```

The `ifs_closeall()` function closes all the files registered with IFS and returns back an integer array populated with the list of the open files that were closed.

This code example is from the `RFSS_suspend()` function in
`sys/src-6400/c6400_remote_filesys.c`.

```

<snip>

/* Close all open files, and receive back a list of which
 * files were open */
ifs_closeall(open_fds);

/* Stop watchdog timers associated with open files */
for (i = 0; (i < IFS_MAX_OPEN_FILES) && open_fds[i]; i++) {
    if (open_fds[i])
        reg_invoke_RFSS_watchdog_stop(open_fds[i]);
}

/* Leave the semaphore locked. The next call to RFSS_resume() should
 * unlock it. */
}

<snip>

```

9.3.2 Other Features

This section describes other features that are supported for the file system's usage. It includes the following subsections:

- Directories
- Timestamps

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.

```
boolean sifs_add_file_to_fs (const fsid_t fsid, const char *directory,
                           sifs_file *new_file);
```

Example 14 - 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 (subsysstype *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 section 9.3.1.3 show a file's modification timestamp being updated.

9.4 Implementing Complete File Systems

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);
typedef int (*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 *prefix,
                                  ino_t ino, struct stat *stat_buf);
typedef int (*ifs_vector_statfs_t)(fsid_t fsid, const char *prefix,
                                   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_ {
    ifs_prefix      *prefix;
    ifs_vector      *vector;
    ifs_flags_t     flags;
    ifs_type_t      type;
    ifs_feature_t   feature;
    int16          num_open;
    void           *fs_context;
    const char     *filename_prompt;
    void           *show_vector;
    ifs_copy_vector *copy_vector;
};


```

The following data structure is the internal representation of a file used by IFS.

```
struct ifs_fdent_ {
    int fd;
    int native_fd;
    fsid_t fsid;
    uint32 filepos;
    uint32 size;
    char *data;
    uint16 flags;
    uint16 oflags;
    pid_t pid;
    int index;
    void *fd_context;

    uint block_count;
    uint block_index;
    uint block_filepos;
    uint block_size;
    char *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 initial 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_RDONLY`, etc., see EDCS-280183, “File Open Flags and their use in IOS Filesystem” for more information on file open flags), 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 whether 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 below:

```
typedef boolean (*ifs_copy_vector_setup_valid_t)(boolean prompt);
typedef boolean (*ifs_copy_vector_check_args_t)
    (void /*ifs_pathent*/ *src_pathent,
     void /*ifs_pathent*/ *dst_pathent);
typedef int     (*ifs_copy_vector_copy_t)
    (void /*ifs_pathent*/ *src_pathent,
     void /*ifs_pathent*/ *dst_pathent, boolean erase,
     boolean verbose);

typedef struct ifs_copy_vector_ {
    ifs_copy_vector_setup_valid_t check_setup;
    ifs_copy_vector_check_args_t  check_args;
    ifs_copy_vector_copy_t       copy;
} 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 patient data structures describing both source and destination file names. If this routine returns FALSE, the copy command is also aborted. The final vector, `copy`, 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 patient 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_t      show_flash_all;
    void ifs_show_flash_chips_t    show_flash_chips;
    void ifs_show_flash_default_t  show_flash_default;
    void ifs_show_flash_detailed_t show_flash_detailed;
    void ifs_show_flash_err_t     show_flash_err;
    void ifs_show_flash_filesys_t show_flash_filesys;
    void ifs_show_flash_summary_t show_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 NVRAM API

The `nvram` file system is used to store the system configuration files: the `startup-config`, containing the visible configuration commands, and the `private-config`, containing non-visible configuration commands.

The NVRAM API can be divided into 3 sections:

- 1 The first set of NVRAM functions are the parser command line interface functions, which are called to write commands and arguments to the `startup-config` and `private-config` configuration files. For more information, see Section 9.6.1, “Parser CLI Interface Functions”.
- 2 The second set of NVRAM functions must be written on a platform-specific basis. For more information, see Section 9.6.2, “Platform-Specific Functions”.
- 3 The third set of NVRAM functions are called by the platform-specific functions in the second set of NVRAM functions, described above. These functions are *not* designed to be called by parser CLI functions. For more information, see Section 9.6.3, “NVRAM Functions Called by Platform-Specific Functions”.

Note All undocumented NVRAM functions should *not* be used without first contacting the file system group via interest-nvram@cisco.com.

9.6.1 Parser CLI Interface Functions

The parser command line interface functions are called to write commands to the startup-config and private-config configuration files.

A variety of NVRAM functions are called to define the start of the command, instruct the parser to augment the current command, or insert the “! ” separator. In addition, you can specify additional work to be performed, such as encrypting the arguments, verifying the length will not exceed the maximum command length, or writing the TTY line number.

The common parser interface functions are listed in Table 9-1, Table 9-2, and Table 9-3.

Table 9-1 nv.h Functions for Parser CLI Support

Function	Description
<code>nv_write_private()</code>	Write configuration strings to RAM as a holding place and after the normal configuration is generated, and re-write the configuration as a separate parse chain in private non volatile storage.
<code>nv_save_private()</code>	Save the private configuration information to NVRAM independently from the information saved by the write mem command.
<code>nv_add_private()</code>	Add a configuration string in private NVRAM as a holding place after the normal configuration is generated.
<code>nv_current_to_new_buffer()</code>	Write the current configuration to a text buffer.
<code>nv_current_to_nv()</code>	Write the current configuration to non volatile storage.
<code>nv_current_to_nv_internal()</code>	Write the current configuration to non volatile storage while printing error messages, supplying file system flags, and requiring confirmation.
<code>nv_write_separator()</code>	Write the ! command separator character between the previous configuration command and the next configuration command in non volatile storage.
<code>nv_interface_function()</code>	Walk the interface parse chain for an IDB.
<code>nv_is_valid()</code>	Check if the non volatile storage is valid.

Table 9-2 nv_common.h Functions for Parser CLI Support

Function	Description
<code>nv_add()</code>	Add to a configuration string in non volatile storage.
<code>nv_add_checklength()</code>	Add a string to NVRAM, first verifying the string is not too long (will not exceed the maximum size of the internal parser buffer).
<code>nv_add_encrypted()</code>	Add DES-encrypted authenticated data to a configuration string in non volatile storage.
<code>nv_write()</code>	Write configuration strings to non volatile storage.

Table 9-2 nv_common.h Functions for Parser CLI Support (continued)

Function	Description
<code>nv_write_checklength()</code>	Write a string to NVRAM, first verifying the string is not too long (will not exceed the maximum size of the internal parser buffer).
<code>nv_write_encrypted()</code>	Write DES-encrypted authentication data to non volatile storage.

Table 9-3 async_chain.c Function for Parser CLI Support

Function	Description
<code>nv_ttyname()</code>	Write a terminal name to the configuration file in non volatile storage.

9.6.2 Platform-Specific Functions

This set of NVRAM functions must be written for each new platform.

For each new platform functions must be defined to support the NVRAM on that platform. For example, to initialize NVRAM, enable and disable access to NVRAM, establish a memory copy of NVRAM, verify the magic number in NVRAM, and update the NVRAM configuration register.

The platform-specific functions are listed in Table 9-4, Table 9-5, and Table 9-6.

Table 9-4 nv.h Platform-Specific Functions

Function	Description
<code>nv_init()</code>	Initialize NVRAM.
<code>nv_badptr()</code>	Check the pointer to the configuration in NVRAM for errors.
<code>nv_writedisable()</code>	Disables writes to NVRAM.
<code>nv_writeenable()</code>	Enables writes to NVRAM.
<code>nv_setconfig()</code>	Write a number to the NVRAM configuration register.
<code>get_nvbase()</code>	Get the pointer to the base of the NVRAM that is usable by the system image.

Table 9-5 nv_common.c Platform-Specific Functions

Function	Description
<code>nv_getptr()</code>	Returns the pointer to the base of NVRAM.
<code>nv_done()</code>	Writes configuration data to NVRAM.

Table 9-6 monitor1.h Platform-Specific Function

Function	Description
<code>platform_nvvar_support()</code>	Return TRUE if the supplied NVRAM variable is supported.

9.6.3 NVRAM Functions Called by Platform-Specific Functions

The third set of NVRAM functions are called by the platform-specific functions described in Section 9.6.2, “Platform-Specific Functions”. These functions are *not* designed to be called by parser CLI functions.

These functions include wrappers around platform-specific functions, providing a clean API, as well as functions returning information about NVRAM.

The generic platform-setup functions are listed in Table 9-7.

Table 9-7 nv.h Functions Called by Platform-Specific Function

Function	Description
<code>nv_check_getptr()</code>	Check the pointer to the configuration in NVRAM before writing the data to NVRAM.
<code>nv_check_done()</code>	Check if the configuration is done before writing the data back to NVRAM.
<code>generic_nv_getptr()</code>	Get the pointer for the configuration saved in NVRAM.
<code>generic_nv_done()</code>	Determine if the configuration is done in NVRAM.
<code>nv_current_uncompressed_size()</code>	Get the current uncompressed size of NVRAM.
<code>ok_to_write_nv()</code>	Verify that it is OK to write to NVRAM.
<code>nv_private_getptr()</code>	Find the address of the private NVRAM storage area.
<code>nv_set_buffered()</code>	Determine if the NVRAM is buffered.
<code>nv_set_valid()</code>	Establish the validity of the NVRAM on the system.

9.7 Persistent Variable Method

The NVRAM scratch space is overpopulated because of the addition of persistent variables and the existing internal interface for Read Only Memory Monitor (ROMMON)-only variables. A scratch space is a small and non-extensible block of memory at the start of NVRAM. This is used for ROMMON-IOS interactions and only accessible in Cisco IOS through an exception call routine (EMT). The EMT also accesses the NVRAM scratch space. All EMT calls lead to the stopping of IOS while the code executes out of ROMMON; this leads to periods of time when interrupts are disabled and packets cannot be switched.

As a result, a new persistent variable method is implemented. The aim is for Cisco IOS to reserve the use of the NVRAM scratch space and the existing internal interface for ROMMON-only variables, such as BOOT. Other IOS-only persistent variables should use an alternative internal interface that hides the storage format and, at an even lower layer, hides the media itself. The new

Persistent Variable Method

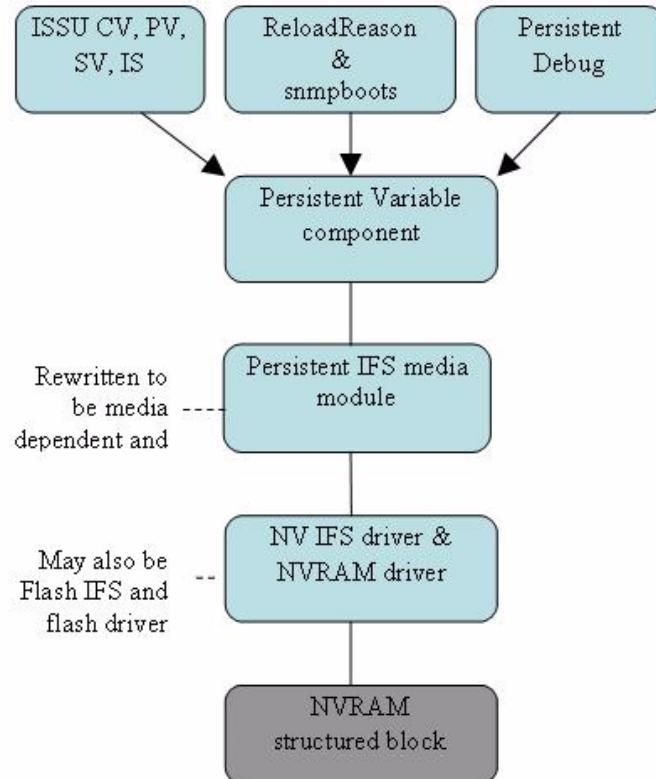
persistent variable method presents a registry API to clients. This registry API hides the internal storage format. The NV IFS persistent module is rewritten to allow an NVRAM-based implementation by default. However, other platforms can use any media, such as flash, and this change does not require the persistent variable module rewrite. The persistent variable module supports features such as caching mechanism, debug, and test CLI support.

The functionality of the persistent variable method is described below:

- Implements a caching mechanism to support fast read access and an optional delayed update of the media upon a write, under the control of the client.
- Uses NVRAM as the default media.
- Provides an ability to replace the default media based on the platform used.
- Maintains the existing NVRAM scratch space method for pure ROMMON variables.
- Moves relevant variables out of the NVRAM scratch space and into the new media without loss of information.
- Implements In Service Software Upgrade (ISSU) as a persistent variable client, thus freeing significant memory in the NVRAM scratch space.
- Resistant to any corruption caused, for example, by a crash during variable write. The solution also handles corruption, such as evidenced in some of the nvram:/persistent data files. In addition, to avoid loss of data during a crash, the design includes a cache flush during the crash exception handling.

9.7.1 Architecture

Figure 9-1 Persistent Data Architecture



In Figure 9-1, the main blocks are the single-sourced persistent variable components. These components support features such as the caching mechanism, hidden variable formatting, and debug and test CLI support. It presents two registry interfaces, an API to its clients, and an API to the media-specific code. The code is a default persistent media module. It interfaces with the IFS driver (NVRAM or flash). If the need is great on a particular platform, the media module and layers below may be substituted, without needing source modification of the layers above.

9.7.1.1 Module Description

This section describes the modules that define the persistent variable feature and the files that require modification.

9.7.1.1.1 Clients

- `os/nv_flash_ifs_chain.c`
Uses the new persistent variable registry API for `ReloadReason` and for the bug fix for `ReloadReason` in the CLI routine.
- `snmp/snmp_nv_ifs_snmpboots.c`
Uses the new persistent variable registry API for `snmpboots`.
- `issu/process/issu_process.c`
`issu/process/issu_process_ipc_agent.c`
`issu/process/issu_process_util.c`

Persistent Variable Method

Uses the new persistent variable registry API for the ISSU variables instead of the `mon_set_var()` and `mon_get_var()` functions.

9.7.1.1.2 Persistent Variable Component

Before single sourcing, all files in the persistent variable component resided in the `/vob/ios/sys/os/persistent/variable` directory with no subdirectories. After a true single source component was created, the component as a whole was moved into the `/vob/ios.comp/persistent/variable` directory with subdirectories (indicated inside the square brace prefix below):

- [include] `prst_vbl_registry.h`

The header file to be included by clients of the persistent variable component. The registry header is bundled with this header file so there is no need to explicitly include it.

- [include/] `prst_vbl_media.h`

The header file to be included by the persistent variable media modules, such as, `prst_ifs.c`. The registry header, `prst_vbl_media_registry.regh`, is bundled with this header, so there is no need to explicitly include it.

- [include/reg] `prst_vbl_registry.c`

Creates the persistent variable registry API during `SUBSYS_CLASS_REGISTRY`.

- [include/reg] `prst_vbl_registry.reg`

The registry API for the interface between the persistent variable component and its clients. See “Persistent Variable Registry Interface” on page 45 for more information.

- [include/reg] `prst_vbl_media_registry.reg`

The registry for the interface between the persistent variable component and the media-specific file, which is by default `/os/prst_ifs.c`. See “Persistent Variable Registry Interface” on page 45 for more information.

- [src/] `prst_vbl.c`

The registry for initialization, core functions (Get, Set), CLI for testing, and debug. It interfaces with the utility module, `src/1 prst_vbl.h`.

- [src/] `prst_vbl.h`

The internal header file for the `[src/] prst_vbl.c` module.

- [src/] `prst_vbl_util.c`

The functions related to the internal format, specifically calls the AVL functions in Cisco IOS to efficiently maintain the persistent variables and their values.

- [src/] `prst_vbl_util.h`

The internal header file for the `[src/] prst_vbl_util.c` module for exclusive use by `prst_vbl.c`.

- [src/] `exec_prst_vbl.h`

The executive command parser definitions.

- [src/] `prst_vbl_chain.c`

The executive command functions.

- [src/] `prst_vbl_chain.h`

The executive command functions header.

- [src/] prst_vbl_debug.c
The debug and error module.
- [src/] prst_vbl_debug.h
The debug and error module header.
- [src/] msg_prst_vbl.c
Error message definitions.

9.7.1.1.3 Persistent Variable High Availability (HA) Subcomponent

The following files allow a local Route Processor (RP) to read, write and flush variables on an Single Sign-On (SSO) peer.

- /ha/src/prst_vbl_ipc_agent.c (h)
IPC connection and send and receive messaging interface.
- /ha/src/prst_vbl_issu_msg.h
ISSU message definitions.
- /ha/src/prst_vbl_issu_transform.c (h)
ISSU registration and transformation module.
- /ha/src/prst_vbl_rf.c (h)
RF progression and status callback functions

9.7.1.1.4 IFS Persistent media module

- os/prst_ifs.c

The IFS persistent media module subsystem initializes during SUBSYS_CLASS_PRE_EHSA, where it registers the routines with the persistent variable media registry. When the media is created, the subsystem kicks off the initialization of the persistent variable component. This subsystem contains the routines that are called from the persistent variable component to read and write to the file using the IFS interface. The IFS persistent media module supports NVRAM by default. Bootflash is also supported. Other file systems are supported; but are untested.

9.7.1.1.5 NV IFS and NVRAM driver changes

- os/nv.h
Removes definitions for nv_ifs_prst_set_string() and similar definitions. These are replaced by the definitions in prst_vbl_registry.reg.
- os/nv_common.c
Changes the class of the nvram_subsys_init subsystem from SUBSYS_CLASS_EHSA to SUBSYS_CLASS_PRE_EHSA to allow high availability (HA) modules to access persistent variables at boot.
- os/nv_ifs.c

Changes the subsystem class from SUBSYS_CLASS_EHSA-“seq: nvram” to SUBSYS_CLASS_PRE_EHSA-“seq: system_ifs”. This allows the HA modules to access the persistent variables at boot. Minor change is needed to this module to allow the software (not via normal CLI) to delete the secure NVRAM persistent data file when requested by the test command. This module is essential for unit testing.

9.7.1.1.6 Makefile Changes

- makesubsys

Creates a new bundle that contains the persistent variable component plus the media-specific module. Creates a new `sub_persistent` subsystem that includes `prst_vbl.o` and `prst_ifs.o`. This subsystem is included in platform builds along with the default, `sub_nv_ifs` and `sub_nv_flash_ifs` (the flash overlay).

9.7.2 Restrictions

The different types of restrictions are described in this section.

9.7.2.1 Software Restrictions and Considerations

In the design where a file is written over the top of an existing one does not take into account the possibility that the file may become corrupt if the system cannot complete the write. For example, if Cisco IOS crashes during the write operation, the file becomes corrupt. It is desirable to avoid this situation by using methods, such as alternatively writing to two files. If one write is interrupted because of a crash, the other file maintains integrity. However, because of the concerns with the NVRAM and NVRAM IFS driver, this functionality is implemented after these drivers are rewritten to correctly support IFS operations.

9.7.2.2 Hardware Restrictions and Considerations

When the driver is performing a buffered write or CRC generation over the entire NVRAM address space, then access to an NVRAM file using the `ifs_open()` and `ifs_close()` functions may be delayed. If the memory size is small, the delay is very minimal. However, for large NVRAM with two megabyte (MB) on a C10K router, this can take more than one second. The driver performs process suspends to avoid CPU hog and is protected against multiple processes accessing the hardware.

9.7.2.3 External Restrictions and Configuration

The following dependencies apply:

- At compile time, the persistent variable component is not dependent on any Cisco IOS code.
- At run time, if the platform invokes the persistent variable API functions, then `shr_prst_vbl_reg.o` should be linked at the platform level to avoid trace backs.
- For the platform to store variables (persistent and nonpersistent), at run time, the `shr_prst_vbl.o` should be linked at the platform level.
- At run time, if the platform expects full persistent data between resets, then the platform should be linked to an IFS module for the media and the media drivers. For most platforms, NVRAM and NVRAM IFS drivers are already linked. For others (including IOU), the flash interface can be used instead. The platform decides this by the return string from a call to `platform_get_string(PLATFORM_STRING_PERSISTENT_PATH)`.

9.7.2.4 Initialization Configuration and Restrictions

There are no initialization configuration requirements. The feature is initialized on its own very early in the initialization sequence (PRE_EHSA) and is available for use during the rest of the initialization period. There is no need for further modification during or after initialization.

Some platforms (generally with older hardware) that do not contain NVRAM or have very limited NVRAM memory, can implement their own media modules to interface with the persistent variables module.

The existing NVRAM driver is designed in a way that it does not lend itself well to the IFS. Because of this, the simplest use of the IFS for the NVRAM is implemented.

9.7.3 Initialization

The initialization of the persistent variable component is independent of the media module and occurs in the SUBSYS_CLASS_PRE_EHSA. This involves registering the routines with the API registry and creating the background processes. In addition, the media module can at any time indicate (via direct call) to the persistent variable component that the media is ready. The persistent variable module can read the information into its cache and run the background process loop waiting for timeouts that request a cache flush.

The internal system flow at initialization of SUBSYS_CLASS_PRE_EHSA is described below:

- 1 Sub System Initialization—prst_vbl_subsys_init()
 - (a) Registers local functions with the API.
 - (b) Creates timers, semaphores, and background processes.
 - (c) Completes initialization if the media indicates that it is ready.
- 2 Finish Initialization—prst_init_finish()
 - (a) Reads the media file and fills the cache.
 - (b) If present in the cache then enables local debugging.
 - (c) If necessary, then performs anticorruption.
 - (d) Initializes CLI for show and debug commands.
 - (e) Persistent variable component is now initialized.
- 3 Background Process—prst_process()
 - (a) If the media is not ready, then suspend until the media is ready.
 - (b) If necessary, then finish initialization.
 - (c) Wait for the rest of system to initialize.
 - (d) If cache has changed, then start the flush timer.
 - (e) Begin the background process loop, waiting for timeouts indicating flush.

9.7.4 Internal Operations

The following operations are supported by the API registry functions:

- **Set**—Sets a persistent variable namevalue string, with a cache or a flush.
- **Is Set**—Determines whether a variable is set (to any value including “”).

- Get—Gets a persistent variable value string.
- Get_Nameval—Gets a persistent variable namevalue string.
- Delete—Gets a persistent variable with a cache or a flush.
- Flush—Flushes all persistent variables to the media, if necessary.

The source flow of the `Set` operation inside `prst_set()` is described below:

- 1 Confirm that the format of the nameval string is as expected.
- 2 Suspend until the semaphore lock is obtained.
- 3 Look into the cache for the name and whether it is present.
- 4 Set the new value, unless the value is the same as the old one.
- 5 If `flush` is requested and the cache is changed, then request a persistent media file flush.
- 6 Release semaphore lock.

The following pseudo-code describes the internal system flow of the `Get` and `Get_Nameval` operations inside `prst_get_inner()`:

- 1 Suspend until the semaphore lock is obtained.
- 2 Look into the cache for the name, if missing release lock and return.
- 3 Confirm that the found value string (or `nameval` if requested) fits into the supplied buffer.
- 4 For the `Get_Nameval` case, copy the name and “=” into the supplied buffer.
- 5 Append the value found into the supplied buffer.
- 6 Release the semaphore lock.
- 7 Return the size of the value in the supplied buffer.

9.7.4.1 Calling the API During Initialization

API calls are not successful if they are called before the persistent variable subsystem is fully initialized. This is because the cache is not read from the media. One exception to this is the `Set-to-cache` function. When the media indicates that it is ready, then the preset cache is merged with data from the media, thus ensuring no information (other than possibly obsolete media data) is lost.

The persistent variable media should be ready before or during the `SUBSYS_CLASS_PRE_EHSA` operation, because clients rely on data to be available. Alternative media that are slow to initialize do not cause any intrinsic errors in the persistent variable component.

After the persistent variable component is initialized, only an explicit flush using `reg_invoke_persistent_flush()` causes a cache flush during initialization.

9.7.4.2 Calling the API Before Reset

API calls read and store variables until the persistent variable “pre-reset” function flushes the data. API calls that do not flush the cache operate without any problems, however further flushes are not done.

9.7.4.3 Calling the API for Disabled Interrupts

All API cache functions called from the process context work, regardless whether the interrupts are enabled or disabled. API flush functions work depending on the media driver. The NVRAM IFS interface allows writes to complete correctly when interrupts are disabled.

Note When interrupts are disabled, persistent debugging to the console is disabled.

9.7.4.4 Calling the API for Interrupt Context

All API cache functions called from the interrupt context work; however, persistent debugging to the console is disabled. All cache-lock logic (using process suspends) is interrupt safe.

API flush functions should not be made during an interrupt handler, because of the delay in writing to the media. After the interrupt handler calls the API to cache the variable, it sets a flag that requests an API flush from a corresponding process.

9.7.5 Cache Flushes

During normal system operation, any API function that modifies the cache results in a cache flush. Whether this occurs immediately (in the caller's process) or after a timeout (in the background process) depends on the flag settings of the API function. The latter is recommended, especially if the cache is modified frequently. The cache is not flushed if the cache is not changed along with the media. In addition, cache flushes also occur when the system is going down.

Local “pre-reset” functions are registered during initialization and handle flushing in the following situations:

- System exception during a system crash.
- Reloads during a normal reload.
- Does switchovers during a normal switchover to a standby RP.

Only an explicit flush using `reg_invoke_persistent_flush()` causes a cache flush during initialization. Flushes do not occur after the “pre-reset” function is executed when the system is going down.

Note Power-offs, hardware crashes, and user aborts because of the break key (signal=0x3 cause=0x20) are not handled and any unflushed data is lost.

9.7.6 Data Structures

This section defines the format of new data structures and extensions or modifications to existing data structures.

9.7.6.1 Cache

An AVL is used for the internal cache. This allows fast variable searches that scale to any number of entries. A structure type is created that includes an AVL node type as the first member and two additional members that point to the name and value information. Wrappers are used around AVL functions to support this node without having to typecast. The node type is defined as:

```
typedef struct {
    avl_node_type avlnode;
    const char *nameval;
    const char *val;
} nameval_node_type;
```

`nameval` points to the stored variable in the same format used by the API. The format is `<name>=<value>'\0'`. `val` points to the value portion within the same string.

9.7.6.2 Handle Context

The media module stores the context for the read and write operation. In the case of the IFS media module, this includes information such as file descriptor (fd), file path, buffer, and buffer pointer information. This structure is referenced by the persistent variable component as a handle of type `prst_vbl_handle` (a `void *`) to avoid including media-specific headers, which it is independent of.

```
typedef struct {
    boolean opened;
    prst_media_open_type open_type;
    int fd;
    const char *filepath;
    /*
     * Temporary buffer to hold file data during reads
     */
    char *buf;
    uint buf_size;
    char *ptr;
    int ptr_len;
} prst_ifs_ctx;
```

9.7.7 Description of Algorithms

Algorithms describe the new and modified code modules. The use of pseudocode, flow charts, and English language descriptions are encouraged. The details should include the following:

- Algorithm(s) used.
- Constraints, dependencies, limitations, or unusual features in the software design.
- Expected conditions of the software when execution begins.
- The conditions under which control is passed to other components.
- Error and exception handling.

9.7.7.1 Cache Memory Allocation

9.7.7.1.1 Initialization

There is no explicit initialization required for the AVL tree. A pointer (of type `nameval_node_type *`) called `top` points to `NULL` and is used internally by all wrapper functions. After the first insert, this pointer points to the first node. As nodes are added and removed, `top` is updated to point always to the top of the AVL tree.

9.7.7.1.2 Insert

After an insert function, a node containing the name is searched using `nameval_search()` (a wrapper of the AVL insert function) to determine if an identical name is already present. If an identical name is present, the existing node is reused—only the string is required for manipulation (that is, either copied if the size is correct, or freed and allocated to the correct size and copied).

If there is no identical name, a `nameval_node_type` is allocated from the heap, as well as a single memory block for both the variable and value (`nameval`). `nameval` is copied into the memory and the node members with `nameval` and `val` are updated to point to the name and value parts, respectively. The `nameval_insert()` function (a wrapper of the AVL insert function) is called to place the node into the tree in alphabetical order of the name part only.

9.7.7.1.3 Delete

Upon a delete, a node containing the name is searched for using `nameval_search()`. If found, the `nameval_delete()` function (a wrapper of the AVL delete function) is called to remove the node from the tree. Then the variable memory is freed and the node is freed.

9.7.8 Internal Variables

The following persistent variables (and example values) are stored by the persistent variable component:

```
prst-ver=1  
prst-fls=123
```

The version variable identifies whether the media file is written to by this feature. The cache is put through a simple anti-corruption algorithm. The flushes variable identifies the number of times the cache is written to the media. This can be used to determine which media file is the newest successfully written file during initialization, reducing file write problems. For more information, see “Software Restrictions and Considerations” on page 36.

The `prst-dbg=1` variable is available when debugging is enabled for the persistent variable module (otherwise it is absent). This is necessary for unit testing.

9.7.9 Persistent Variable Media Module

This module sits between the persistent variable component and the persistent media. By default, the module is available in the `/os/prst_ifs.c` directory, which uses the IFS interface. This accesses a file called “`nvram:persistent-data`”. The prefix and filename can be overridden by the platform using `platform_get_string(PLATFORM_STRING_PERSISTENT_PATH)`. Persistent storage by using other media (such as flash) or the filename is supported by the platform. This is done by overriding the string as given in Table 9-8:

Table 9-8 Platform_get_string information

Platform_get_string (PLATFORM_STRING_PERSISTENT_PATH)	Persistent Path
NULL (the default)	“nvram:persistent-data”
“:<file>”	“nvram:<file>”
“<media>:”	“<media>:persistent-data”
“<media>:<file>”	“<media>:<file>”

The persistent variable media module ensures that the IFS driver subsystems are initialized. This module ensures that NVRAM or other media is initialized in the first release. The media module can be absent in an image link. However, this means that data is stored in a non-persistent manner.

This module ensures the integrity of the stored data if this is not enforced by the underlying driver. For example, the NVRAM driver performs a CRC over the complete address space any time data is written. So no further action is required in the default case.

This module initializes its subsystem in the SUBSYS_CLASS_PRE_EHSA class. The initialization routine registers a number of its functions with `prst_vbl_media_registry.reg`.

The functions registered for the read and write functions are described below:

- `handle = prst_ifs_open(PRST_MEDIA_READ)`
Opens the persistent file (using `ifs_open`) and determines the size (`ifs_fd_get_size`). Allocates a memory buffer of this size and reads the file into this buffer (`ifs_read_until`). The file is then closed (`ifs_close`) and points to the current position at the start of the buffer. The context information is stored in a single static structure and the handle to this structure is returned.
- `prst_ifs_read(handle, const char **nameval)`
Correctly formats one name and value in the buffer according to the current position of the context. The pointer to this name and value is returned. The caller has to obtain a copy of this data.
- `prst_ifs_close(handle)`
Frees the buffer and nulls the current position.
- `handle = prst_ifs_open(PRST_MEDIA_WRITE)`
Opens (`ifs_open`) and empties the persistent file.
- `prst_ifs_write (handle, const char *nameval)`
Appends (`ifs_write`) one name and value in the correct format to the persistent file.
- `prst_ifs_close(handle)`
Closes the persistent file (`ifs_close`).

Other routines registered with the Media Registry Interface on page 47 are:

- `prst_media_show()`
- `prst_media_get_next_open_path()`
- `prst_media_delete()`

This module also has the capability to:

- Perform timing of the routines mentioned above. These times are displayed along with other information when debugging is enabled.
- Deleting the secure file (nvram:persistent-data) for the purposes of testing.

The following ROMMON variables are added to the software to provide support for the bring-up testing of different media types. These are not a unit-testable facility and should not be set by anyone other than for developers.

- PRST_VBL_DEBUG=1
 - Enables early debugging that includes IFS initialization and creation.
- PRST_IFS_PREFIX=<filesystem>:
 - Overrides the default (or platform selected) filesystem.

9.7.10 HA Client

Persistent variables are clients of HA, specifically RF and ISSU. The registration and messaging (one specific message per API) is unexceptional. After the successful ISSU negotiation of the message types, the local system (active or standby) allows access to the peer variables through the API by using the following flags:

- PERSISTENT_PEER_RP—for get, set, isset, delete and flush affecting only the peer.
- PERSISTENT_BOTH_RP—for set, delete, and flush affecting both local and peer.

9.7.11 Interface Design

The interface design section defines the interfaces between the modules and the external interfaces used by other computer programs and hardware. This section includes interface parameter information such as:

- parameter name
- purpose
- data type
- size and format
- range or enumeration of values (for example, 0-99)

Interface design includes any security and privacy considerations as well as any dependencies on other product interfaces. It also identifies the sources (that is, the setting or sending entities) and the recipients (that is, the using or receiving entities) for each interface.

9.7.11.1 Makefile Changes

Currently the cross platform and platform-specific subsystem structure is as follows:

```
makefile:  
NV_IFS = shr_nv_ifs.o shr_nv_ifs_util.o  
  
makesubsys:  
sub_nv_ifs = nv_ifs.o nv_ifs_registry.o \  
            ifs_nvram_format_routines.o \  
            ifs_nvram_blockio_routines.o \  
            nv_ifs_prst.o snmp_nv_ifs_snmpboots.o
```

Persistent Variable Method

The changes prior to single sourcing are:

```
/vob/ios/sys/makefile:
    PRST_VBL = shr_prst_vbl_reg.o shr_prst_vbl.o prst_ifs.o

    OS_CORE =  shr_core.o shr_tty.o sub_core_platform.o shr_bb_debug.o \
               $(SCHED_INTERVAL) $(EVENT_TRACE) $(MEMORY_COMPONENT) \
               ...
               $(EVENT_MGR) $(IDMGR) $(PRST_VBL)

/vob/ios/sys/makesubsys:
    sub_prst_vbl_desc = "Persistent Variable Client Registry"
    sub_prst_vbl_reg =      prst_vbl_registry.o

    sub_prst_vbl_desc = "Persistent Variable Component"
    sub_prst_vbl =        prst_vbl.o prst_vbl_util.o \
                        prst_vbl_chain.o prst_vbl_debug.o

    sub_nv_ifs = nv_ifs.o nv_ifs_registry.o \
                 ifs_nvram_format_routines.o \
                 ifs_nvram_blockio_routines.o \
                 snmp_nv_ifs_snmpboots.o
```

The changes once the component is able to be single-sourced are:

```
/vob/cisco.com/persistent/variable/src.mak:
    SOURCES.comp_prst_vbl.o := \
        prst_vbl_registry.o \

    SOURCES.comp_prst_vbl.o := \
        prst_vbl.o \
        prst_vbl_util.o \
        prst_vbl_chain.o \
        prst_vbl_debug.o

/vob/ios/sys/makesubsys:
    sub_nv_ifs = nv_ifs.o nv_ifs_registry.o \
                 ifs_nvram_format_routines.o \
                 ifs_nvram_blockio_routines.o \
                 snmp_nv_ifs_snmpboots.o
```

All platforms that link to the OS_CORE fully support this feature down to the media-specific IFS module. A platform can override this in the makefile by redefining the PRST_VBL function.

9.7.11.2 Persistent Variable Registry Interface

At a source file level, any client can use the persistent variable registry API by including the <prst_vbl/include/prst_vbl_registry.h> file. This header automatically brings in the registry header so there is no need to include it explicitly. The following API is defined below. An example of usage is given in “Source Code” on page 50.

```
#  
# Each of these registry functions take a parameter of type persistent_flags.  
# This field modifies the behavior of the call in the following way:  
#  
# PERSISTENT_DEFAULT Default behavior. That is, access variables on this  
# (the local) RP, and for gets, get the value.  
# PERSISTENT_PEER_RP Variables on the peer RP are modified or inspected,  
# PERSISTENT_BOTH_RPS Variables on both local and peer RP are modified.  
# <other flags> Some API functions have additional flags, eg. Get.  
# These are listed under the registry function comment.  
#  
# Each of these registry functions, upon error, will return a code of  
negative  
# value of type persistent_rc. The codes common to all functions that can be  
# returned are:  
#  
# PERSISTENT_RC_NOTREADY The API can not yet/no longer be used,  
# PERSISTENT_RC_PARAM A parameter has an illegal value,  
# PERSISTENT_RC_PEER The peer cannot be communicated with,  
# PERSISTENT_RC_LOCK There is a cache lock problem,  
# <other codes> Some API functions will return other return codes.  
  
#  
#  
# DEFINE persistent_set  
/*  
 * Write a new (or replace if existing) persistent variable.  
 * The string passed in as nameval can be of the format "<name>=[<val>]".  
 * eg "<name>=" is acceptable, but "<name>" by itself is not.  
 *  
 * Upon success, returns the length (incl terminator) of nameval.  
 * Upon failure, returns one of the common return codes (all < 0), or...  
 * PERSISTENT_RC_CACHE if there is a cache problem during the set.  
 */  
    STUB_CHK  
    persistent_rc  
    persistent_flags flags, const char *nameval  
END  
  
# DEFINE persistent_is_set  
/*  
 * Inspect the cache to find the persistent variable pointed to by name.
```

Persistent Variable Method

```

/*
 * Upon success, returns size of the found value (incl terminator).
 * Upon failure, returns one of the common return codes (all < 0), or...
 *   PERSISTENT_RC_MISSING if the variable is missing in the cache.
 */
    STUB_CHK
    persistent_rc
    persistent_flags flags, const char *name
END

DEFINE persistent_get
/*
 * Inspect the cache to find the persistent variable pointed to by name.
 * If val is not NULL and the variable is found and the string size to write
 * is less than maxsize, then it is copied into the buffer pointed to by val.
 * The persistent_flags has the following additional options:
 *   PERSISTENT_NAMEVAL to copy both name and value of form <name>=<value>.
 *
 * Also val may point inside the name buffer as long as
 * there is room for the value to be written.
 *
 * Upon success, returns the size of the string written into val.
 * Upon failure, returns one of the common return codes (all < 0), or...
 *   PERSISTENT_RC_MISSING if the variable is missing in the cache,
 *   PERSISTENT_RC_TOOBIG if the string to be written
 *   (including terminator) is larger than maxsize
 */
    STUB_CHK
    persistent_rc
    persistent_flags flags, const char *name, char *val, uint maxsize
END

DEFINE persistent_delete
/*
 * Delete the persistent variable pointed to by name. Both name and
 * any value is removed. Does nothing if the variable is not present.
 *
 * Upon success, returns...
 *   PERSISTENT_RC_OK.
 * Upon failure, returns one of the common return codes (all < 0), or...
 *   PERSISTENT_RC_CACHE if there is a cache problem while deleting.
 */
    STUB_CHK
    persistent_rc
    persistent_flags flags, const char *name
END

DEFINE persistent_flush
/*
 * Update the underlying media from the internal cache so that it
 * contains the latest persistent variables and values.
 * Does nothing if update is not necessary (but still treated as a success).
 *
 * Upon success, returns...
 *   PERSISTENT_RC_OK.
 * Upon failure, returns one of the common return codes (all < 0), or...
 *   PERSISTENT_RC_MEDIA if there is problem updating the underlying media.
 */
    STUB_CHK

```

```

    persistent_rc
    persistent_flags flags
END

```

9.7.11.2.1 Virtual Switch Support

The support for a virtual switch will not impact the persistent variable registry interface. The client specifies the PERSISTENT_BOTH_RPS parameter in the flag. On a virtual switch-enabled platform the persistent variable infrastructure sets the variable on the local RP, SSO peer, and the in-chassis peer. If either or both peers do not exist, the local persistent variable infrastructure marks those peers as requiring a sync and performs the sync once these come into existence.

9.7.11.3 Media Registry Interface

The registry interface between the persistent variable component and the media-specific file is given below. By default, the `persistent_ifs.c` registers these service points and then directly calls the persistent variable initialization routine. From that point on, the persistent variable component (and only that component) invokes these routines.

Note Only one media-specific module can install its functions with this registry at any one time. An example of usage is given in “Source Code” on page 50.

```

DEFINE prst_media_open
/*
 * Called from the persistent variable component to request
 * the media module be ready for persistent variable reading
 * or writing, depending on the type parameter.
 * Returns handle to the context, or 0 upon failure.
 */
STUB_CHK
prst_media_handle
prst_media_open_type open_type
END

DEFINE prst_media_read
/*
 * Called from the persistent variable component to read one variable
 * from the media module. It is up to the media module to allocate the
 * buffer and copy the variable and value into this buffer.
 * The address of this buffer is returned in *nameval and the format
 * of the buffer is "<name>=<val>". Upon return the buffer is the
 * responsibility of the persistent variable component.
 * Returns PRST_MEDIA_ERROR upon failure,
 * PRST_MEDIA_EMPTY read was not sucessful there are no more
 * variables available to get.
 * PRST_MEDIA_OK otherwise.

```

Persistent Variable Method

```

    */
    STUB_CHK
    prst_media_rc
    prst_media_handle handle, const char **nameval
END

DEFINE prst_media_write
/*
 * Called from the persistent variable component to write one variable
 * (in nameval format eg. "<name>=<val>") to the media module.
 * Returns PRST_MEDIA_ERROR upon failure,
 * PRST_MEDIA_OK otherwise.
*/
    STUB_CHK
    prst_media_rc
    prst_media_handle handle, const char *nameval
END

DEFINE prst_media_close
/*
 * Called from the persistent variable component to indicate to the
 * media module that variable reading or writing is complete.
 * Returns PRST_MEDIA_ERROR upon failure,
 * PRST_MEDIA_OK otherwise.
*/
    STUB_CHK
    prst_media_rc
    prst_media_handle handle
END

DEFINE prst_media_show
/*
 * Show the content of the persistent media.
*/
    STUB_CHK
    void
-
END

DEFINE prst_media_get_next_open_path
/*
 * Return a pointer to the path name of the media entry to
 * be read or written in the next call to prst_media_open()
*/
    STUB_CHK
    const char *
-
END

DEFINE prst_media_delete
/*
 * Delete a persistent media entry (eg. file).
 * Upon failure, return < 0, otherwise return 0.
*/
    STUB_CHK
    int
    const char *media_del
END

```

9.7.12 Memory and Performance Impact

To store a copy of the persistent data, it is estimated that several new modules and a RAM-based cache are required. The persistent data itself is stored as an IFS NVRAM file by most platforms and resides in the available NVRAM block memory. The amount of persistent memory used depends on the variables stored; however, it is estimated to be around 500 bytes for the initial release supporting the following persistent variables:

The memory for persistent variables are listed in Table 9-9.

Table 9-9 Memory for Persistent Variables

Persistent Variable	Size in Bytes
Two Overhead variables (version plus a counter)	25 X 2
Reload Reason	22 + 15 (est. for reason string)
SNMP boots	15
ISSU variables	20 (est) x 3 (CV,PV,SV) + 5 (IS)
Five (approximate) debugs	60 (est) x 5
TOTAL	500

It has been suggested that the EMT call for a write to NVRAM scratch space takes between 170 milliseconds (MS) and 300 MS to complete, during which time, there is no Cisco IOS processing (including interrupts). With the new mechanism, the performance of the underlying media is critical for a read (during initialization) and when the media is flushed (at any time, depending on the client). Specifically, this relates to the performance of NVRAM and the NVRAM IFS on a per-platform basis. The table below attempts to compare the expected performance of the two methods.

The NVRAM performance for both the methods are listed in Table 9-10.

Table 9-10 NVRAM Performance for Both Methods

Platform	NVRAM Size	EMT Variable Write	NVRAM IFS Table Flush	NVRAM IFS Table Read
MSFC3	Absent			
3620	30 KB			<100 MS (estimated)
7200	128 KB		60 MS	
GSR	507 KB	170-300 ms	200 MS (estimated)	
7300 (NPEG100-800)	1 MB		324 MS	<5 MS
c10K	2 MB		1300 MS	500 MS

- EMT variable write times have not been confirmed.
- NVRAM IFS table access times have been measured for the following sequences: `ifs_open()`, `ifs_read()`, `ifs_write()`, or `ifs_close()`. The numbers with the postfix are estimates based on the observations using the CLI `copy` and `dir` commands upon NVRAM.
- NVRAM IFS allows interrupts to occur and it may suspend other processing during IFS calls.

- Future changes to the architecture of the NVRAM may mean some large numbers will drop significantly (for example the complete NVRAM isn't copied and/or CRC'ed for each file write).
- The use of intelligent caching by clients will reduce this overhead significantly.

9.7.13 Source Code

The following example displays the Get and Set API calls, as used by clients of the persistent variable component:

```
#include <prst_vbl/include/prst_vbl_registry.h>

#define SNMP_MAX_SNMPBOOTS_DIGITS 10
#define SNMP_SNMPBOOTS_VAR "snmpboots"

char value_buf[SNMP_MAX_SNMPBOOTS_DIGITS + 1];
persistent_rc ret_code = reg_invoke_persistent_get(PERSISTENT_DEFAULT,
                                                    SNMP_SNMPBOOTS_VAR, value_buf,
                                                    sizeof(value_buf));
if (ret_code <= 0) {
    snmpboots = 0;
} else {
    snmpboots = atoi(value_buf);
}

#include <prst_vbl/include/prst_vbl_registry.h>
#define SNMP_SNMPBOOTS_VAR "snmpboots"

char name_value_pair[SNMP_MAX_SNMPBOOTS_NAME_VALUE_PAIR + 1];

persistent_rc ret_code;
snprintf(name_value_pair, SNMP_MAX_SNMPBOOTS_NAME_VALUE_PAIR + 1,
         "%s=%d", SNMP_SNMPBOOTS_VAR, snmpboots);
ret_code = reg_invoke_persistent_set(PERSISTENT_DEFAULT,
                                      name_value_pair, PERSISTENT_FLUSH);
if (ret_code <= 0) {
    return FALSE;
}
return TRUE;
```

The following code is taken from the persistent variable component and shows the media read operation that refreshes the cache. The media write operation has a similar structure.

```
#include "prst_vbl_media.h"

/*
 * Refresh the cache from the media.
 * Returns TRUE if successful, FALSE otherwise.
 */
boolean prst_util_cache_refresh (void)
{
    boolean rc;
    const char *nameval;
    prst_media_handle handle;

    /*
     * Three steps are involved here according to the registry interface:
     * 1.media_open() 2.multiple media_read() calls 3.media_close().
     */
```

```
/*
handle = reg_invoke_prst_media_open(PRST_MEDIA_READ);
if (!handle) {
    prst_buginf("Cache Refresh - failed to open");
    return FALSE;
}
do {
    rc = reg_invoke_prst_media_read(handle, &nameval);
    if (rc != PRST_MEDIA_OK) {
        prst_buginf("Cache Refresh - media read %s", nameval);
        break;
    }
    if (!prst_util_insert(nameval)) {
        prst_buginf("Cache Refresh - nameval insert %s", nameval);
        rc = PRST_MEDIA_ERROR;
        break;
    }
} while (rc == PRST_MEDIA_OK);
if (reg_invoke_prst_media_close(handle) != PRST_MEDIA_OK) {
    prst_buginf("Cache Refresh - failed to close");
    return FALSE;
}
return TRUE;
```

9.8 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.

NVRAM MULTIPLE FILE SYSTEM: Software Unit Design/Functional Specification for the NVRAM Multiple File System for IOS. NVRAM provides storage for configuration and persistent data files. EDCS-66111

<http://wwwwin-eng.cisco.com/Eng/IOSTech/IOSInf/classic-nvram.doc@latest>

RSP Flash File System Design Document. EDCS- ENG-4829.

IFS Functional Specification. EDCS- ENG-10472.

IFS Design Specification. EDCS-ENG-10473

File System Analysis Tool.

IOS File System:Copying Files.

IOS File System: DOS Formatting.

LFN Specification and API Functions. EDCS-182754.

Filesystem SNMP MIB Query: How-Tos. EDCS-241982.

Static Datastructures in the PCMCIAFS ATA Filesystem. EDCS-182756.

Remote File System.

Using the Flash Disk

IOS File System Commands

IOS PCMCIAFS Test CLI

References

Persistent Variables—Development Test Detailed Test Plan EDCS-487826

Persistent Variable—Software Design and Unit Test Specification EDCS-465563

Persistent Variable—Software Functional Specification EDCS-464461

Socket Interface

This chapter lists the functions available in the Cisco IOS socket interface. The `**_event **()` named functions are part of the socket event delivery mechanism which is proprietary to Cisco. The rest of the Cisco IOS socket interface functions implement a subset of the standard UNIX socket functions and perform identically or almost identically to their counterparts in the Berkeley UNIX socket library.

The Cisco IOS socket interface functions work over TCP and UDP. With a few exceptions (see Table 10-1), the function names are the same as those in the standard UNIX socket library plus `socket_` has been added to the beginning of each name.

Note Socket code development questions can be directed to the `interest-socket@cisco.com` alias.

Table 10-1 lists the socket function calls supported by the Cisco IOS socket implementation, along with their standard UNIX equivalent if available.

Table 10-1 Cisco IOS Socket Functions and Macros

Function Name	Standard UNIX Function Name
<code>if_indextoname()</code>	None
<code>if_outputindex()</code>	None
<code>inet_ntop()</code>	None
<code>process_get_socket_event()</code>	None
<code>process_set_socket_interest()</code>	None
<code>process_set_socket_interest_all()</code>	None
<code>socket_accept()</code>	<code>accept()</code>
<code>socket_allocbuf_recvfrom()</code>	None
<code>socket_bind()</code>	<code>bind()</code>
<code>socket_close()</code>	<code>close()</code>
<code>socket_connect()</code>	<code>connect()</code>
<code>socket_get_localname()</code>	<code>getsockname()</code>
<code>socket_get_max_sockets()</code>	None
<code>socket_get_option()</code>	<code>getsockopt()</code>
<code>socket_get_peername()</code>	<code>getpeername()</code>

Table 10-1 Cisco IOS Socket Functions and Macros (continued)

Function Name	Standard UNIX Function Name
<code>socket_gethostbyname()</code>	<code>gethostbyname()</code>
<code>socket_inet_addr()</code>	<code>inet_addr()</code>
<code>socket_inet_network()</code>	<code>inet_network()</code>
<code>socket_inet_ntoa()</code>	<code>inet_ntoa()</code>
<code>socket_inherit_fd()</code>	None
<code>socket_inherit_fds()</code>	None
<code>socket_listen()</code>	<code>listen()</code>
<code>socket_open()</code>	<code>socket()</code> and <code>open()</code>
<code>socket_recv()</code>	<code>recv()</code>
<code>socket_recvfrom()</code>	<code>recvfrom()</code>
<code>socket_recvmsg()</code>	<code>recvmsg()</code>
<code>socket_select()</code>	<code>select()</code>
<code>socket_send()</code>	<code>send()</code>
<code>socket_sendmsg()</code>	<code>sendmsg()</code>
<code>socket_sendto()</code>	<code>sendto()</code>
<code>socket_set_max_sockets()</code>	None
<code>socket_set_option()</code>	<code>setsockopt()</code>
<code>socket_share_fds()</code>	None
<code>socket_shutdown()</code>	<code>shutdown()</code>
<code>socket_strerror()</code>	<code>strerror()</code>
<code>socket_watch_other_events()</code>	None
Macro Name	
<code>FD_CLR</code>	<code>FD_CLR</code>
<code>FD_ISSET</code>	<code>FD_ISSET</code>
<code>FD_SET</code>	<code>FD_SET</code>
<code>FD_SETSIZE</code>	<code>FD_SETSIZE</code>
<code>FD_ZERO</code>	<code>FD_ZERO</code>

For information about the standard socket interface, refer to the following publications:

- Richard Stevens, *UNIX Network Programming*, Volume I, Third Edition (currently available with your CEC password at: <http://cisco.safaribooksonline.com>)
- Douglas Comer, *Internetworking with TCP/IP*, Volumes I and III
- Richard Stevens and Gary Wright, *TCP/IP Illustrated*, Volumes 1 and 2 (currently available with your CEC password at: <http://cisco.safaribooksonline.com>)

10.1 SCTP Sockets API

The SCTP (Stream Control Transmission Protocol) sockets API functions available in Cisco IOS have been updated (*New in 12.4T*) to support a new event notification mechanism, the new `MSG_NOTIFICATION` socket message flag. The new/updated API functions are the preferred API for SCTP sockets for new applications. All applications should use the UNIX style sockets functions available in IOS as the main API to SCTP and `IPPROTO_SCTP` should be used as the protocol when opening a new socket.

The new/updated SCTP sockets API provides support for both the one-to-many model (the preferred use of SCTP) using the `SOCK_SEQPACKET` type, as well as a one-to-one model (primarily for a seamless TCP migration path) using the `SOCK_STREAM` type. The following UNIX style sockets functions in IOS have been modified to support the one-to-one and one-to-many SCTP models (see the API reference pages, such as `socket_bind()` in the *Cisco IOS API Reference* for more information on SCTP support):

- 1 `socket_accept()`
- 2 `socket_bind()`
- 3 `socket_close()`
- 4 `socket_connect()`
- 5 `socket_get_localname()`
- 6 `socket_get_option()`
- 7 `socket_get_peername()`
- 8 `socket_listen()`
- 9 `socket_open()`
- 10 `socket_recv()`
- 11 `socket_recvfrom()`
- 12 `socket_recvmsg()`
- 13 `socket_send()`
- 14 `socket_sendmsg()`
- 15 `socket_sendto()`
- 16 `socket_set_option()`
- 17 `socket_shutdown()`

10.1.1 New SCTP Sockets API Functions

The following new functions are defined by the SCTP sockets API:

- 1 `sctp_bindx()`
- 2 `sctp_getpaddrs()`
- 3 `sctp_freepaddrs()`
- 4 `sctp_getladdrs()`
- 5 `sctp_freeladdrs()`
- 6 `sctp_sendmsg()`
- 7 `sctp_recvmsg()`
- 8 `sctp_connectx()`

```

9 sctp_send()
10 sctp_sendx()
11 sctp_peeloff()
12 sctp_opt_info()

```

The user should include "socket/sctp_uio.h" to use these new SCTP API functions. The reference documentation for the new SCTP sockets API functions is in chapters 9, 10, and 23 of *UNIX Network Programming*, Volume I, Third Edition (currently available with your CEC password at: <http://cisco.safaribooksonline.com>).

10.1.2 Cisco-Specific SCTP Sockets Options

The following new SCTP sockets options are currently Cisco-specific and are *not* defined in the SCTP sockets API:

- 1** SCTP_CONTEXT—Set a context value on the socket that the user can use.
- 2** SCTP_DELAYED_ACK_TIME—Set the delayed SACK value.
- 3** SCTP_UNORDERED_PRIORITY—Set the priority for unordered data (equal or high).
- 4** SCTP_MAXBURST—Set the maximum burst value.
- 5** SCTP_AUTO_ASCONF—Enable/disable automatic ASCONF generation. Deferred.
- 6** SCTP_MAX_ASSOC—The maximum number of associations per socket.
- 7** SCTP_RESET_STREAMS—Deferred until the SCTP Reset Streams feature is committed.
- 8** SCTP_GET_ASOC_ID_LIST—Get the list of association IDs on a given socket (this is a read-only option).
- 9** SCTP_GET_NONCE_VALUES—Set the V-tag nonce values for an association (this is a read-only option).
- 10** SCTP_GET_PEGS—Deferred (this is a read-only option).
- 11** SCTP_GET_STAT_LOG—Deferred (this is a read-only option).

For more information on the implementation of SCTP support in IOS, see EDCS-434293.

10.1.3 SCTP One-to-One Code Example

The following code is a simple implementation of an echo server over SCTP. The example shows how to use some features of one-to-one model IPv4 SCTP sockets, including:

- Opening, binding, and listening for new associations on a socket.
- Enabling ancillary data.
- Enabling notifications.
- Using ancillary data with `socket_sendmsg()` and `socket_recvmsg()`.
- Using `MSG_EOR` to determine if an entire message has been read.
- Handling notifications.

```
#include "../socket/socket.h"
#include "../socket/sctp_uio.h"

#define BUFSIZE 100
#define INET6_ADDRSTRLEN 100

static void
handle_event (void *buf)
{
    struct sctp_assoc_change *sac;
    struct sctp_send_failed *ssf;
    struct sctp_paddr_change *spc;
    struct sctp_remote_error *sre;
    union sctp_notification *snp;
    char addrbuf[INET6_ADDRSTRLEN];
    const char *ap;
    sockaddr_in *sin;
    sockaddr_in6 *sin6;

    snp = buf;
    switch (snp->sn_header.sn_type) {
    case SCTP_ASSOC_CHANGE:
        sac = &snp->sn_assoc_change;
        printf("\n^^^ assoc_change: state=%u, error=%u, instr=%u outstr=%u",
               sac->sac_state, sac->sac_error, sac->sac_inbound_streams,
               sac->sac_outbound_streams);
        break;
    case SCTP_SEND_FAILED:
        ssf = &snp->sn_send_failed;
        printf("\n^^^ sendfailed: len=%u err=%d", ssf->ssf_length,
               ssf->ssf_error);
        break;
    case SCTP_PEER_ADDR_CHANGE:
        spc = &snp->sn_paddr_change;
        if (spc->spc_aaddr.ss_family == AF_INET) {
            sin = (sockaddr_in *)&spc->spc_aaddr;
            ap = inet_ntop(AF_INET, &sin->sin_addr, addrbuf,
                           INET6_ADDRSTRLEN);
        } else {
            sin6 = (sockaddr_in6 *)&spc->spc_aaddr;
            ap = inet_ntop(AF_INET6, &sin6->sin6_addr, addrbuf,
                           INET6_ADDRSTRLEN);
        }
        printf("\n^^^ intf_change: %s state=%d, error=%d", ap,
               spc->spc_state, spc->spc_error);
        break;
    case SCTP_REMOTE_ERROR:
        sre = &snp->sn_remote_error;
        printf("\n^^^ remote_error: err=%u len=%u",
               ntohs(sre->sre_error), ntohs(sre->sre_length));
        break;
    case SCTP_SHUTDOWN_EVENT:
        printf("\n^^^ shutdown event");
        break;
    default:
        printf("\nunknown type: %u", snp->sn_header.sn_type);
        break;
}
```

```

        }

    }

    static void *
mysctp_recvmsg (int fd, struct msghdr *msg, void *buf, size_t *buflen,
                ssize_t *nrp, size_t cmsglen)
{
    ssize_t nr = 0, nnr = 0;
    struct iovec iov[1];

    *nrp = 0;
    iov->iov_base = buf;
    iov->iov_len = *buflen;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;
    for (;;) {
#ifndef MSG_XPG4_2
#define MSG_XPG4_2 0
#endif
        msg->msg_flags = MSG_XPG4_2;
        msg->msg_controllen = cmsglen;
        nnr = socket_recvmsg(fd, msg, 0);
        if (nnr <= 0) {
            /* EOF or error */
            *nrp = nr;
            return (NULL);
        }
        nr += nnr;
        if ((msg->msg_flags & MSG_EOR) != 0) {
            *nrp = nr;
            return (buf);
        }
        /* Realloc the buffer? */
        if (*buflen == (size_t)nr) {
            buf = realloc(buf, *buflen * 2);
            if (buf == 0) {
                buginf("\nout of memory");
                exit(1);
            }
            *buflen *= 2;
        }
        /* Set the next read offset */
        iov->iov_base = (char *)buf + nr;
        iov->iov_len = *buflen - nr;
    }
}

static void
echo (int fd, int socketModeone_to_many)
{
    ssize_t nr;
    struct sctp_sndrcvinfo *sri;
    struct msghdr msg[1];
    struct cmsghdr *cmsg;
    char cbuf[sizeof (*cmsg) + sizeof (*sri)];
    char *buf;
    size_t buflen;
    struct iovec iov[1];
    size_t cmsglen = sizeof (*cmsg) + sizeof (*sri);
}

```

```
/* Allocate the initial data buffer */
buflen = BUFLEN;
if (!(buf = malloc(BUFLEN))) {
    buginf("\nout of memory");
    exit(1);
}
/* Set up the msghdr structure for receiving */
memset(msg, 0, sizeof (*msg));
msg->msg_control = cbuf;
msg->msg_controllen = cmsglen;
msg->msg_flags = 0;
cmsg = (struct cmsghdr *)cbuf;
sri = (struct sctp_sndrcvinfo *)(cmsg + 1);
/* Wait for something to echo */
while (buf = mysctp_recvmsg(fd, msg, buf, &buflen, &nr, cmsglen))
{
    /* Intercept notifications here */
    if (msg->msg_flags & MSG_NOTIFICATION) {
        handle_event(buf);
        continue;
    }
    iov->iov_base = buf;
    iov->iov_len = nr;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;
    printf("\ngot %u bytes on stream %u:", nr, sri->sinfo_stream);

    /* Echo it back */
    msg->msg_flags = MSG_XPG4_2;
    if (socket_sendmsg(fd, msg, 0) < 0) {
        socket_strerror("sendmsg");
        exit(1);
    }
}
if (nr < 0) {
    socket_strerror("recvmsg");
}
if (socketModeone_to_many == 0)
    socket_close(fd);
}

int
one_to_one_echo_main (void)
{
    struct sctp_event_subscribe event;
    int lfd, cfd;
    sockaddr_in sin[1];

    if ((lfd = socket_open(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1)
    {
        socket_strerror("socket");
        exit(1);
    }
    sin->sin_family = AF_INET;
    sin->sin_port = htons(7);
    sin->sin_addr.s_addr = INADDR_ANY;
    if (socket_bind(lfd, (sockaddr *)sin, sizeof (*sin)) == -1)
    {
```

```

        socket_strerror("bind");
        exit(1);
    }
    if (socket_listen(lfd, 1) == -1) {
        socket_strerror("listen");
        exit(1);
    }
    /* Wait for new associations */
    for (;;) {
        if ((cfid = socket_accept(lfd, NULL, 0)) == -1) {
            socket_strerror("accept");
            exit(1);
        }
        /* Enable all events */
        event.sctp_data_io_event = 1;
        event.sctp_association_event = 1;
        event.sctp_address_event = 1;
        event.sctp_send_failure_event = 1;
        event.sctp_peer_error_event = 1;
        event.sctp_peer_error_event = 1;
        event.sctp_shutdown_event = 1;
        event.sctp_partial_delivery_event = 1;
        event.sctp_adaption_layer_event = 1;
        if (socket_set_option(cfid, IPPROTO_SCTP, SCTP_EVENTS, &event,
                             sizeof(event)) != 0) {
            socket_strerror("setevent failed");
            exit(1);
        }
        /* Echo back any and all data */
        echo(cfid, 0);
    }
}
}

```

10.1.4 SCTP One-to-Many Code Example

The following code is a simple implementation of an echo server over SCTP. The example shows how to use some features of one-to-many model IPv4 SCTP sockets, including:

- Opening and binding a socket.
- Enabling ancillary data.
- Enabling notifications.
- Handling notifications.

```

#include "../socket/socket.h"
#include "../socket/sctp_uio.h"

int
one_to_many_echo_main (void)
{
    int fd;
    int idleTime = 2;
    sockaddr_in sin[1];
    struct sctp_event_subscribe event;

    if ((fd = socket_open(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) == -1) {
        socket_strerror("socket");
    }

```

```

        exit(1);
    }
    sin->sin_family = AF_INET;
    sin->sin_port = htons(7);
    sin->sin_addr.s_addr = INADDR_ANY;
    if (socket_bind(fd, (sockaddr *)sin, sizeof (*sin)) == -1)
    {
        socket_strerror("bind");
        exit(1);
    }

    /* Enable all notifications and events */
    event.sctp_data_io_event = 1;
    event.sctp_association_event = 1;
    event.sctp_address_event = 1;
    event.sctp_send_failure_event = 1;
    event.sctp_peer_error_event = 1;
    event.sctp_shutdown_event = 1;
    event.sctp_partial_delivery_event = 1;
    event.sctp_adaption_layer_event = 1;
    if (socket_set_option(fd, IPPROTO_SCTP, SCTP_EVENTS, &event,
                         sizeof(event)) != 0) {
        socket_strerror("setevent failed");
        exit(1);
    }

    /* Set associations to auto-close in 2 seconds of inactivity */
    if (socket_set_option(fd, IPPROTO_SCTP, SCTP_AUTOCLOSE, &idleTime, 4) <
        0) {
        socket_strerror("setsockopt SCTP_AUTOCLOSE");
        exit(1);
    }
    /* Allow new associations to be accepted */
    if (socket_listen(fd, 1) < 0) {
        socket_strerror("listen");
        exit(1);
    }
    /* Wait for new associations */
    while (1) {
        /* Echo back any and all data */
        echo(fd, 1);
    }
}

```

10.2 Useful Commands

To find out all the socket ports that are open on an IOS router, enter the **show control-plane host open-ports** command (*New in 12.4T*). For example, here is sample output from this command:

Active internet connections (servers and established)				
Prot	Local Address	Foreign Address	Service	State
tcp	*:23	*:0	Telnet	LISTEN
tcp	*:80	*:0	HTTP CORE	LISTEN
udp	10.4.22.18:56641	*:0	IP SNMP	LISTEN
udp	10.4.22.18:162	*:0	IP SNMP	LISTEN
udp	10.4.22.18:161	*:0	IP SNMP	LISTEN
udp	10.4.22.18:67	*:0	DHCPD Receive	LISTEN

Useful Commands

This sample output only shows TCP and UDP ports over IPv4. For ephemeral ports such as those initiated from the router (tftp/ssh/telnet), the state displays “ESTABLISHED”.

Standard Libraries

This chapter underwent a complete overhaul because of the new security guidelines. (December 2010)

Note Because Cisco has adopted the Cisco Secure Development Lifecycle (CSDL), which is a mandatory process to improve the security of Cisco software, this chapter has undergone a complete overhaul. Wherever possible, notes about security issues have been embedded. Follow the recommendations with great care in order to make Cisco code more secure.

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work.

11.1 Before the Security Initiative

Starting with Release 11.2, the Cisco IOS software began formalized use of ANSI C library functions. More recent releases brought efforts to provide support for POSIX standard library functions, some of which overlap with traditional ANSI C library functions. Standard library function support was then generalized to improve the portability of Cisco IOS software. Standard library functions were ported to the Cisco IOS environment as needed, tested for standards compliance, and integrated into the mainline source trains.

The *Cisco IOS API Reference, Chapter 11, “Standard Libraries,”* lists the POSIX and ANSI C library functions that have been ported and conform to the above standards. See also Table 11-3, “Available ANSI C, POSIX, and Related Library Functions,” in this chapter. For functions that are compliant and have no notable API differences, the Cisco documentation does not duplicate the standards reference information. Rather, for reference information on these functions, see the official standards documentation links in Section 11.4, “Standard Library Support,” in this chapter.

The *Cisco IOS API Reference* includes reference pages for Cisco-specific versions or variations of standard library functions. Also included are some reference pages for legacy Cisco IOS standard ANSI C functions that were previously documented, although they might not differ from standard support.

Note There is no support in the Cisco IOS software for floating-point operations, and there are no plans to support floating-point operations.

At the time of writing, the current version of the GNU CC compiler (GCC) used for Cisco IOS development is GCC 3.4.

For information about GCC, see:

<http://gcc.gnu.org/onlinedocs/gcc/index.html>

The EDCS document for GCC 3.4/GDB 6.1 is:

EDCS-383144

Note Standard library development questions can be directed to compiler-dev@cisco.com.

11.2 Security Initiative

The number of headlines regarding denial of service attacks, network outages, and other hacker exploits has grown significantly year over year. With the increasing frequency and virulence of the attacks, product security is more important than ever before. As a market leader, Cisco is expected to deliver resilient products that can withstand attacks. Our customers look to us to enable their networks to be safe and secure, and expect product security to be seamlessly integrated into the broad range of products we sell. In order to do this, we must all increase our understanding of secure product development. Further, we must integrate this knowledge into our product architecture, design, and development processes so that product security becomes part of our DNA and corporate culture.

At this time, the Cisco Secure Development Lifecycle (CSDL), a repeatable and measurable process designed to mitigate the risk of vulnerabilities and increase resiliency of Cisco products in the face of attacks, is mandatory for secure products. The CSDL Safe Library element specifies a collection of secure libraries for C, C++, and Java-based products. A developer on the team, typically a tech lead, is responsible for integrating the safe library into the code base. All developers on the team must use safe functions when developing new code.

How is the security initiative related to standard libraries?

A number of standard C lib functions are prone to buffer overflows (for example, `strcpy()`, `memcpy()`, and so on), and serve as launch points for more sophisticated attacks. The safer C library functions (TR 24731 and rationale document N1147) defined by the ISO C committee are a set of “safe” replacements to standard C lib functions. The Safe C specification has gone through full ISO approval and is part of the C Standard. When used properly, these functions decrease the danger of buffer overrun attacks. The ISO/IEC rationale behind TR 24731 is as follows:

- Guard against overflowing a buffer.
- Do not produce unterminated strings.
- Do not unexpectedly truncate strings.
- Provide a library useful to existing code.
- Preserve the `NULL`-terminated string datatype.
- Only require local edits to programs.

- Provide a library-based solution.
- Support compile-time checking.
- Make failures obvious.
- Provide zero buffers and NULL strings.
- Provide a run-time-constraint handler mechanism.
- Support reentrant code.
- Have a consistent naming scheme.
- Have a uniform pattern for the function parameters and return type.
- Have a deference to existing technology.

Cisco's Safe C implementation is derived from the ISO/IEC TR 24731 specification.

11.2.1 Mapping Between Unsafe Functions and Safe Functions

A number of standard C library functions are prone to buffer overflows (for example, `strcpy()`, `memcpy()`, and so on), and serve as launch points for more sophisticated attacks. The safer C library functions is a set of safe replacements to standard C lib functions. In all cases, the engineer should use the safe library functions rather than the standard and unsafe functions. Table 11-1 provides a mapping between unsafe memory functions and safe memory functions.

Table 11-1 Mapping Between Unsafe Memory Functions and Safe Memory Functions

Standard and Unsafe Functions	Description	Safe Library Function Replacement
<code>bcopy()</code>	<code>bcopy()</code> copies a source region to a destination region.	<code>memcpy_s()</code>
<code>bzero()</code>	<code>bzero()</code> zeros a block of memory.	<code>memzero_s()</code>
<code>memcmp()</code>	<code>memcmp()</code> compares two regions in byte units returning indication of less than, equal, or greater than.	<code>memcmp_s()</code>
<code>memcpy()</code>	<code>memcpy()</code> copies bytes from the source region to the destination region.	<code>memcpy_s()</code>
<code>memmove()</code>	<code>memmove()</code> copies in byte units, allowing overlap, from the source region to the destination region.	<code>memmove_s()</code>
<code>memrchr()</code>	<code>memrchr()</code> finds a character in a region.	None
<code>memset()</code>	<code>memset()</code> sets all bytes in the region to the given value.	<code>memset_s()</code>

Table 11-2 provides a mapping between unsafe string functions to safe string functions.

Table 11-2 Mapping Between Unsafe String Functions and Safe String Functions

Standard and Unsafe Functions	Description	Safe Library Function Replacement	Similar Safe C Function Replacement
cmpid()	cmpid() compares two strings for length bytes. The comparison does <i>not</i> stop when a '\0' byte is detected.	strcmpfld_s()	—
concat()	concat() concatenates two strings to create a third string.	Use: strcat_s()(s1, s1max, s2) strcat_s()(s1, s1max, s3)	—
lowercase()	lowercase() converts a string to lowercase.	strtolowercase_s()	—
remove_ws()	remove_ws() removes leading and trailing white spaces in the given string.	strremovews_s()	—
strcasecmp()	strcasecmp() compares the source string with the destination string in a case-insensitive manner.	strcasecmp_s()	—
strcasestr()	strcasestr() performs a case-insensitive strstr(), locating the first occurrence in the source string of the sequence of characters in the destination string.	strcasestr_s()	—
strcat()	strcat() concatenates the source string to the destination string.	strcat_s()	—
strchr()	strchr() returns a pointer to the first occurrence of the c character in the destination string.	strfirstchar_s(), strfirstdiff_s()	—
strcmp()	strcmp() compares the source string with the destination string.	strcmp_s(), strcasecmp_s(), strcmpfld_s()	—
strcpy()	strcpy() copies the source string to the destination string.	strcpy_s(), strcpyfld_s(), strcpyfldin_s(), strcpyfldout_s()	—
strcspn()	strcspn() searches the dest string for characters that are <i>not</i> in the source string.	strcspn_s()	—

Table 11-2 Mapping Between Unsafe String Functions and Safe String Functions (continued)

Standard and Unsafe Functions	Description	Safe Library Function Replacement	Similar Safe C Function Replacement
strcmp()—Not documented in this chapter.	strcmp() returns an offset to the first different character between the source and destination strings.	strfirstdiff_s()	—
isalnum()	isalnum() verifies that a character is alphanumeric.	None	strisalphanumeric_s() —Verifies that a string is alphanumeric.
isascii()	isascii() verifies that a character is ASCII.	None	strisascii_s()—Verifies that a string is ASCII.
isdigit()	isdigit() verifies that a character is decimal digits.	None	strisdigit_s()—Verifies that a string is decimal digits.
isxdigit()	isxdigit() verifies that a character is hexadecimal.	None	strishex_s()—Verifies that a string is hexadecimal.
islower()	islower() verifies that a character is lowercase.	None	strislowercase_s()—Verifies that a string is lowercase.
isupper()	isupper() verifies that a character is uppercase.	None	strisuppercase_s()—Verifies that a string is uppercase.
strlcat()	strlcat() appends a source string to a destination string of a certain size, assuring the final string is NULL-terminated.	strcat_s()	—
strlcpy()	strlcpy() copies a string of a certain size from a source string to a destination string, assuring the final string is NULL-terminated and no buffer overflow occurs.	strcpy_s()	—
strlen()	strlen() returns the string length.	strnlen_s()	—
strncat()	strncat() appends up to n characters from the source string to the destination string and then appends a terminating NULL character.	strncat_s()	—

Table 11-2 Mapping Between Unsafe String Functions and Safe String Functions (continued)

Standard and Unsafe Functions	Description	Safe Library Function Replacement	Similar Safe C Function Replacement
strcmp()	strcmp() compares not more than <i>n</i> bytes from the source string to the destination string.	strcmp_s()	—
strncpy()	strncpy() copies not more than <i>n</i> bytes from the source string to the destination string. If copying takes place between objects that overlap, the behavior is undefined.	strncpy_s()	—
strpbrk()	strpbrk() locates the first occurrence in the destination string of any byte from the source string.	strpbrk_s()	—
strrchr()	strrchr() returns a pointer to the last occurrence of the <i>c</i> character in the destination string.	strlastchar_s(), strlastdiff_s()— returns the index of the last character that is different between the source and destination strings.	—
strspn()	strspn() computes the length of the maximum initial segment of the destination string that consists entirely of characters from the source string.	strspn_s()	—
sstrncat()	sstrncat() is Cisco's safe version of strncpy(). Note that it always NULL-terminates the string if the source is greater than or equal to the maximum length. In these situations, the ANSI strncpy() would copy the string up to the maximum length and <i>not</i> NULL-terminate the string.	strspn_s()	—

Table 11-2 Mapping Between Unsafe String Functions and Safe String Functions (continued)

Standard and Unsafe Functions	Description	Safe Library Function Replacement	Similar Safe C Function Replacement
sstrncpy()	sstrncpy() is Cisco's safe version of strncpy(). Note that it always NULL-terminates the string if the source is greater than or equal to the maximum length. In these situations, ANSI strncpy() would copy the string up to the maximum length and <i>not</i> terminate the string.	strncpy_s()	—
uppercase()—Not documented in this chapter.	uppercase() converts a string to uppercase.	strtouppercase_s()	—
strstr()	strstr() finds a string segment.	strstr_s(), strcasestr_s()	—
strtok	strtok() retrieves the next token from a string.	strtok_s()	—

11.3 Safe Library Functions

The following sections describe the safe library functions:

- Handling Run-time Constraints
- Comparing Values in Memory
- Copying a Region of Memory
- Copying a Block of Memory, Handling the Overlap
- Initializing an Area of Memory to a Desired Value
- Zeroing Bytes
- Comparing Strings by Converting Them to Uppercase
- Locating the First Occurrence (Case-sensitive) of a Substring
- Appending a Copy of a String
- Comparing Two Strings
- Comparing Two Character Arrays
- Copying a String
- Copying from One Character Array to Another
- Copying Characters from a String to an Array
- Copying Characters from an Array to a String
- Counting the Number of Characters That Are Not in a String
- Retrieving a Pointer to the First Occurrence of a Character in a String

- Retrieving the Index of the First Character That Is Different Between Two Strings
- Retrieving the Index of the First Character That Is the Same Between Two Strings
- Checking Whether the Entire String Contains Alphanumeric Characters
- Checking Whether the Entire String Contains ASCII
- Checking Whether the Entire String Contains Digits
- Checking Whether the Entire String Contains Hexadecimal Characters
- Checking Whether the Entire String Is Lowercase
- Checking Whether the Entire String Is Mixed Case
- Validating the Makeup of a Password String
- Checking Whether the Entire String Is Uppercase
- Retrieving a Pointer to the Last Occurrence of a Character in a String
- Retrieving the Index of the Last Character That Is Different Between Two Strings
- Retrieving the Index of the Last Character That Is the Same Between Two Strings
- Removing the Beginning White Space from a String by Shifting the Text Left
- Concatenating Two Strings
- Copying a Counted Nonoverlapping String
- Computing the Length of a String
- Retrieving a Pointer to the First Occurrence of Any Character Contained in One String That Is Also Contained in Another String
- Determining Whether the Prefix Pointed to by One String Is at the Beginning of Another String
- Removing Beginning and Trailing White Spaces From a String
- Computing the Prefix Length of a String
- Locating the First Occurrence of a Substring Regardless of Case
- Retrieving the Next Token from a String
- Scanning the String Converting Uppercase Characters to Lowercase Characters
- Scanning the String Converting Lowercase Characters to Uppercase Characters
- Nulling a String

11.3.1 Handling Run-time Constraints

To handle a run-time constraint, call the `invoke_safe_str_constraint_handler()` function.

```
#include "safe_mem_constraint.h"
invoke_safe_str_constraint_handler(const char *msg,
                                    void *ptr,
                                    errno_t error);
```

11.3.2 Comparing Values in Memory

To compare values in memory, call the `memcmp_s()` function.

```
#include "safe_mem_lib.h"
errno_t memcmp_s(const void *dest, rsize_t dmax, const void *src, rsize_t
slen, int *diff);
```

11.3.3 Copying a Region of Memory

To copy a region of memory, call the `memcpy_s()` function.

```
#include "safe_mem_lib.h"
errno_t memcpy_s(void *dest, rsize_t dmax, const void *src, rsize_t slen);
```

11.3.4 Copying a Block of Memory, Handling the Overlap

To copy a block of memory, handling the overlap, call the `memmove_s()` function.

```
#include "safe_mem_lib.h"
errno_t memmove_s(void *dest, rsize_t dmax, const void *src, rsize_t slen);
```

11.3.5 Initializing an Area of Memory to a Desired Value

To initialize an area of memory to a desired value, call the `memset_s()` function.

```
#include "safe_mem_lib.h"
errno_t memset_s(void *dest, rsize_t len, uint8 value);
```

11.3.6 Zeroing Bytes

To zero `len` bytes starting at `dest`, call the `memzero_s()` function.

```
#include "safe_mem_lib.h"
errno_t memzero_s(void *dest, rsize_t len);
```

11.3.7 Comparing Strings by Converting Them to Uppercase

To compare strings by converting them to uppercase prior to the comparison, call the `strcasectcmp_s()` function.

```
#include "safe_str_lib.h"
errno_t strcasectcmp_s(const char *dest, rsize_t dmax, const char *src, int
*indicator);
```

11.3.8 Locating the First Occurrence (Case-sensitive) of a Substring

To locate the first occurrence of a substring, call the `strcasestr_s()` function.

```
#include "safe_str_lib.h"
errno_t strcasestr_s(char *dest, rsize_t dmax, const char *src, rsize_t slen,
char **substring);
```

11.3.9 Appending a Copy of a String

To append a copy of a string, call the `strcat_s()` function.

```
#include "safe_str_lib.h"
errno_t strcat_s(char *dest, rsize_t dmax, const char *src);
```

11.3.10 Comparing Two Strings

To compare two strings, call the `strcmp_s()` function.

```
#include "safe_str_lib.h"
errno_t strcmp_s(const char *dest, rsize_t dmax, const char *src, int
*indicator);
```

11.3.11 Comparing Two Character Arrays

To compare two character arrays, call the `strcmpfld_s()` function.

```
#include "safe_str_lib.h"
errno_t strcmpfld_s(const char *dest, rsize_t dmax, const char *src, int
*indicator);
```

11.3.12 Copying a String

To copy a string, use the `strcpy_s()` function.

```
#include "safe_str_lib.h"
errno_t strcpy_s(char *dest, rsize_t dmax, const char *src);
```

11.3.13 Copying from One Character Array to Another

To copy from one character array to another, call the `strcpyfld_s()` function.

```
#include "safe_str_lib.h"
errno_t strcpyfld_s(char *dest, rsize_t dmax, const char *src, rsize_t slen);
```

11.3.14 Copying Characters from a String to an Array

To copy characters from a string to an array, call the `strcpyfldin_s()` function.

```
#include "safe_str_lib.h"
errno_t strcpyfldin_s(char *dest, rsize_t dmax, const char *src, rsize_t
slen);
```

11.3.15 Copying Characters from an Array to a String

To copy characters from an array to a string, call the `strcpyfldout_s()` function.

```
#include "safe_str_lib.h"
errno_t strcpyfldout_s(char *dest, rsize_t dmax, const char *src, rsize_t
slen);
```

11.3.16 Counting the Number of Characters That Are Not in a String

To count the number of characters that are not in a string, call the `strcspn_s()` function.

```
#include "safe_str_lib.h"
errno_t strcspn_s(const char *dest, rsize_t dmax, const char *src, rsize_t
slen, rsize_t *count);
```

11.3.17 Retrieving a Pointer to the First Occurrence of a Character in a String

To retrieve a pointer to the first occurrence of a character in a string, call the `strfirstchar_s()` function.

```
#include "safe_str_lib.h"
errno_t strfirstchar_s(char *dest, rsize_t dmax, char c, char **first);
```

11.3.18 Retrieving the Index of the First Character That Is Different Between Two Strings

To retrieve the index of the first character that is different between strings, call the `strfirstdiff_s()` function.

```
#include "safe_str_lib.h"
errno_t strfirstdiff_s(const char *dest, rsize_t dmax, const char *src,
rsize_t *index);
```

11.3.19 Retrieving the Index of the First Character That Is the Same Between Two Strings

To retrieve the index of the first character that is the same between two strings, call the `strfirstsame_s()` function.

```
#include "safe_str_lib.h"
errno_t strfirstsame_s(const char *dest, rsize_t dmax, const char *src,
rsize_t *index);
```

11.3.20 Checking Whether the Entire String Contains Alphanumeric Characters

To check whether the entire string contains alphanumeric characters, call the `strisalphanumeric_s()` function.

```
#include "safe_dest_lib.h"
boolean strisalphanumeric_s(const char *dest, rsize_t dmax);
```

11.3.21 Checking Whether the Entire String Contains ASCII

To check whether the entire string contains ASCII, call the `strisascii_s()` function.

```
#include "safe_str_lib.h"
boolean strisascii_s(const char *dest, rsize_t dmax);
```

11.3.22 Checking Whether the Entire String Contains Digits

To check whether the entire string contains digits, call the `strisdigit_s()` function.

```
#include "safe_str_lib.h"
boolean strisdigit_s(const char *dest, rsize_t dmax);
```

11.3.23 Checking Whether the Entire String Contains Hexadecimal Characters

To check whether the entire string contains hexadecimal characters, call the `strishex_s()` function.

```
#include "safe_str_lib.h"
boolean strishex_s(const char *dest, rsize_t dmax);
```

11.3.24 Checking Whether the Entire String Is Lowercase

To check whether the entire string is lowercase, call the `strislowercase_s()` function.

```
#include "safe_str_lib.h"
boolean strislowercase_s(const char *dest, rsize_t dmax);
```

11.3.25 Checking Whether the Entire String Is Mixed Case

To check whether the entire string is mixed case, call the `strismixedcase_s()` function.

```
#include "safe_str_lib.h"
boolean strismixedcase_s(const char *str, rsize_t dmax);
```

11.3.26 Validating the Makeup of a Password String

To validate the makeup of a password string, call the `strispASSWORD_s()` function.

```
#include "strlib.h"
boolean strispASSWORD_s(const char *dest, rsize_t dmax);
```

11.3.27 Checking Whether the Entire String Is Uppercase

To check whether the entire string is uppercase, call the `strisuppercase_s()` function.

```
#include "safe_str_lib.h"
boolean strisuppercase_s(const char *dest, rsize_t dmax);
```

11.3.28 Retrieving a Pointer to the Last Occurrence of a Character in a String

To retrieve a pointer to the last occurrence of a character in a string, call the `strlastchar_s()` function.

```
#include "safe_str_lib.h"
errno_t strlastchar_s(char *dest, rsize_t dmax, char c, char **last);
```

11.3.29 Retrieving the Index of the Last Character That Is Different Between Two Strings

To retrieve the index of the last character that is different between two strings, call the `strlastdiff_s()` function.

```
#include "safe_str_lib.h"
errno_t strlastdiff_s(const char *dest, rsize_t dmax, const char *src,
rsize_t *index);
```

11.3.30 Retrieving the Index of the Last Character That Is the Same Between Two Strings

To retrieve the index of the last character that is the same between two strings, call the `strlastsame_s()` function.

```
#include "safe_str_lib.h"
errno_t strlastsame_s(const char *dest, rsize_t dmax, const char *src,
rsize_t *index);
```

11.3.31 Removing the Beginning White Space from a String by Shifting the Text Left

To remove the beginning white space from a string by shifting the text left, call the `strljustify_s()` function.

```
#include "safe_str_lib.h"
errno_t strljustify_s(char *dest, rsize_t dmax);
```

11.3.32 Concatenating Two Strings

To concatenate two strings, call the `strncat_s()` function.

```
#include "safe_str_lib.h"
errno_t strncat_s(char *dest, rsize_t dmax, const char *src, rsize_t slen);
```

11.3.33 Copying a Counted Nonoverlapping String

To copy a counted nonoverlapping string, call the `strncpy_s()` function.

```
#include "safe_str_lib.h"
errno_t strncpy_s(char *dest, rsize_t dmax, const char *src, rsize_t slen);
```

11.3.34 Computing the Length of a String

To compute the length of the string pointed to by another string, call the `strnlen_s()` function.

```
#include "safe_str_lib.h"
rsize_t strnlen_s(const char *dest, rsize_t dmax);
```

11.3.35 Retrieving a Pointer to the First Occurrence of Any Character Contained in One String That Is Also Contained in Another String

To retrieve a pointer to the first occurrence of any character contained in one string that is also contained in another string, call the `strpbrk_s()` function.

```
#include "safe_str_lib.h"
errno_t strpbrk_s(char *dest, rsize_t dmax, char *src, rsize_t slen, char **first);
```

11.3.36 Determining Whether the Prefix Pointed to by One String Is at the Beginning of Another String

To determine whether the prefix pointed to by one string is at the beginning of another string, call the `strprefix_s()` function.

```
#include "safe_str_lib.h"
```

11.3.37 Removing Beginning and Trailing White Spaces From a String

To remove beginning and trailing white spaces from a string, call the `strremovews_s()` function.

```
#include "safe_str_lib.h"
errno_t strremovews_s(char *dest, rsize_t dmax);
```

11.3.38 Computing the Prefix Length of a String

To compute the prefix length of a string pointed to by another string, call the `strspn_s()` function.

```
#include "safe_str_lib.h"
errno_t strspn_s(const char *dest, rsize_t dmax, const char *src, rsize_t selen, rsize_t *count);
```

11.3.39 Locating the First Occurrence of a Substring Regardless of Case

To locate the first occurrence (regardless of case) of the substring pointed to by `src` that would be located in the string pointed to by `dest`, call the `strstr_s()` function.

```
#include "safe_str_lib.h"
errno_t strstr_s(char *dest, rsize_t dmax, const char *src, rsize_t selen, char **substring);
```

11.3.40 Retrieving the Next Token from a String

To retrieve the next token from a string, call the `strtok_s()` function.

```
#include "safe_str_lib.h"
char * strtok_s(char *dest, rsize_t *dmax, char *src, char **ptr);
```

11.3.41 Scanning the String Converting Uppercase Characters to Lowercase Characters

To scan the string converting uppercase characters to lowercase characters, call the `strtolowercase_s()` function.

```
#include "safe_str_lib.h"
errno_t strtolowercase_s(char *dest, rsize_t dmax);
```

11.3.42 Scanning the String Converting Lowercase Characters to Uppercase Characters

To scan the string converting lowercase characters to uppercase characters, call the `strtouppercase_s()` function.

```
#include "safe_str_lib.h"
errno_t strlouppercase_s(char *dest, rsize_t dmax);
```

11.3.43 Nulling a String

To NULL a string, call the `strzero_s()` function.

```
#include "safe_str_lib.h"
errno_t strzero_s(char *dest, rsize_t dmax);
```

11.4 Standard Library Support

This section gives some background information about Cisco IOS standard library function support for the following:

- ANSI C Library Functions
- POSIX Library Functions

11.4.1 ANSI C Library Functions

ANSI C has been superseded by ISO/IEC 9899 and is now an international standard. The International Electrotechnical Commission (IEC) is a global organization that prepares and publishes international standards for electrical, electronic, and related technologies. The following publications on the international standard C Programming Language are available for purchase at <http://www.iec.ch/index.html>:

- ISO/IEC 9899 (1999-12) C Programming Language
- ISO/IEC 9899 Corr.1(2000-09) C TECHNICAL CORRIGENDUM 1

11.4.1.1 ANSI C Standard Header Files

In intermediate releases before the most current 12.2S and 12.4T trains:

In Cisco IOS software, ANSI C standard library function declarations and supporting definitions are in the following include files:

- `stdlib.h` for ANSI C standard function definitions.

- `string.h` for ANSI C string function definitions.
- `sys/lib/ansi/include/errno.h` for ANSI C standard error codes.
- `sys/lib/cisco/include/ciscolib.h` for prototypes of Cisco-specific implementations of ANSI C functions `inlibcisco.a`.

In release trains 12.2S, 12.4T, and later:

In Cisco IOS, declarations for ANSI C standard library functions are in the `stdlib.h` include file in the POSIX standard library function source tree. For function prototype declarations, use the following definition to automatically use the proper include file with operating system-specific mappings:

```
COMP_INC(posix, stdlib.h)
```

11.4.2 POSIX Library Functions

Many POSIX library functions have been ported to Cisco IOS 12.2S, 12.4T, and later release trains. These are based on and compliant with the following standard:

POSIX IEEE-1003.1-2001 (ANSII)

Table 11-3, “Available ANSI C, POSIX, and Related Library Functions,” in this section lists the ANSI C library functions and IEEE-1003.1-2001 POSIX standard library functions that have been ported to Cisco IOS mainline source trains.

Use the following guidelines to locate functions you are interested in using:

- Function Grouping and Links:

Functions are grouped by type of function or related library grouping, and listed alphabetically within each group, as follows:

- Math Functions (`stdlib.h`)
- Stdlib, Other Miscellaneous Functions (`ctype.h`, `errno.h` `stdlib.h`, `unistd.h`)
- File System and Stdio Functions (`dirent.h`, `stdio.h`, `unistd.h`)
- String and Memory Copy Functions (`string.h`; `cisco/include/ciscolib.h <standards extension function declarations>`; includes local functions)
- Signal Functions (`signal.h`)
- Semaphore Functions (`semaphore.h`, `sys/sem.h`)
- Socket Functions (`sys/socket.h`): Implemented in Release 12.2S/12.4T as calls to Cisco IOS `socket_xxx_equivalent` functions with limited POSIX standard conformance
- Time Functions (`time.h`, `sys/time.h`, `sys/times.h`)
- Timer Functions (`time.h`, `sys/time.h`)

Links are provided to the reference pages for nonconforming or legacy functions that are included later in this chapter. Functions listed without active links do not have reference pages in this chapter because they conform to the standard. To access reference pages for usage details on these functions, see section 11.4.2.211.4.2.2, “Accessing POSIX Standards Reference Documentation”.

- Header File Inclusion:

When using standard functions, include the appropriate POSIX/ANSI C standard header files for each function using the `COMP_INC(posix, ...)` macro, which automatically includes the proper header file with operating system-specific mappings.

- How Compliant/Noncompliant Implementations Coexist in Cisco IOS Library Source:

Cisco IOS software has some previously developed noncompliant versions of POSIX function names still in use in some platforms whose names conflict with the new, POSIX-compliant implementations. To resolve the conflict until noncompliant versions are deprecated, the Cisco IOS POSIX-compliant function names take the form:

`posix_<standard-POSIX-function-name>()`

Note The string and memory functions in Table 11-3, in the Description, Compliance and Other Notes column have a note indicating whether the function is an unsafe function and, if so, what the safe function is.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions

Function Name	Function Prototype	Description, Compliance, and Other Notes
Math Functions (stdlib.h)		
<code>abs()</code>	<code>int abs(int i)</code>	Integer absolute value. ANSI C standard.
<code>atoi()</code>	<code>int atoi(const char *s)</code>	Convert string to integer. ANSI C standard.
<code>atol()</code>	<code>long int atol(const char *s)</code>	Convert string to long integer. ANSI C standard.
<code>div()</code>	<code>div_t div(int numer, int denom)</code>	Integer divide (quotient, remainder). ANSI C standard.
<code>labs()</code>	<code>long int labs(int i)</code>	Long integer absolute value. ANSI C standard.
<code>ldiv()</code>	<code>ldiv_t ldiv(int numer, int denom)</code>	Long integer divide (quotient, remainder). ANSI C standard.
Stdlib, Other Miscellaneous Functions (ctype.h, errno.h stdlib.h, unistd.h)		
<code>_ctype_</code>	<code>const unsigned char _ctype_[]</code>	Character type classification table for character type classification API functions and macros. POSIX standard (can be locale-dependent) implementation in <code>kernel/libc/ctype</code> . Platform-specific macro implementations also exist in Cisco IOS software. For portability, use the standard <code>libc</code> version if possible.
<code>assert()</code>	<code>void assert(scalar expression)</code>	Various platform-specific implementations; not tested for ANSI C/POSIX standard conformance.
<code>calloc()</code>	<code>void *calloc(size_t nmemb, size_t size)</code>	Allocate unused memory space, initialized to all zeros. ANSI C standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
errno	Macro returns current errno dereferenced integer value.	Error return value. ANSI C standard.
free()	void *free(void *ptr)	Free allocated memory space. ANSI C standard.
getcwd()	char* getcwd(char *buf, size_t size)	Get current working directory pathname. ANSI C standard.
gethostname()	int gethostname(char *name, size_t len)	Get name of current host. ANSI C standard, but depends on local include file sys/os/hostname.h.
getlogin()	char *getlogin(void)	Get user login name associated with the terminal session of the calling process. ANSI C standard.
getpid()	pid_t getpid(void)	Get the process ID of the calling process. ANSI C standard.
isalnum()	int isalnum(int c)	Test for an alphanumeric character. ANSI C standard; previously implemented, conformance to be checked.
isalpha()	int isalpha(int c)	Test for an alphabetic character. ANSI C standard; previously implemented, conformance to be checked.
isascii()	int isascii(int c)	Test whether an integer is an ASCII character. POSIX standard implementation (ANSI C extension function) in kernel/libc. Platform-specific macro implementations also exist in Cisco IOS software. For portability, use the standard libc version if possible.
iscntrl()	int iscntrl(int c)	Test for a control character. ANSI C standard; previously implemented, conformance to be checked.
isdigit()	int isdigit(int c)	Test for a decimal digit character. ANSI C standard; previously implemented, conformance to be checked.
isgraph()	int isgraph(int c)	Test for a visible character. ANSI C standard; previously implemented, conformance to be checked.
islower()	int islower(int c)	Test for a lowercase letter. ANSI C standard; previously implemented, conformance to be checked.
isprint()	int isprint(int c)	Test for a printable character. ANSI C standard; previously implemented, conformance to be checked.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
ispunct()	int ispunct(int c)	Test for a punctuation character. ANSI C standard; previously implemented, conformance to be checked.
isspace()	int isspace(int c)	Test for a white-space character. ANSI C standard; previously implemented, conformance to be checked.
isupper()	int isupper(int c)	Test for an uppercase letter. ANSI C standard; previously implemented, conformance to be checked.
isxdigit()	int isxdigit(int c)	Test for a hex digit. ANSI C standard; previously implemented, conformance to be checked.
longjmp()	void longjmp(jmp_buf env, int val)	Non-local transfer of control. Platform-specific implementations currently; not tested for ANSI C/POSIX standard conformance.
malloc()	void *malloc(size_t size)	Allocate available memory areas. Conforms to ANSI C standard, <i>except Cisco IOS malloc() and related memory allocation functions zero the allocated space.</i>
qsort()	void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *))	Sorts an array of data. Platform-specific implementations currently; not tested for ANSI C/POSIX standard conformance.
rand()	int rand(void)	Pseudorandom number sequence generator. Various Cisco IOS platform-specific implementations of this ANSI C function, with the baseline 1ibc version currently mapping to the Cisco IOS (noncrypto-secure) random_gen() utility function, which has number generation in the range 1—10,000, whereas ANSI C/POSIX standard specifies number generation in the range 0 to ((2 ³¹)—1).
random()	long int random(void)	Returns a series of pseudorandom numbers on successive calls. Current implementation simply invokes Cisco IOS rand() (not ANSI C/POSIX standard compliant).
realloc()	void *realloc(void *ptr, size_t size)	Change the size of a specified block and return a pointer to the (possibly moved) block. ANSI C standard, but Cisco IOS malloc() behavior of returning zero-filled memory results when <i>ptr</i> is NULL.
setjmp()	int setjmp(jmp_buf env)	Set the jump point for nonlocal control transfer. Platform-specific implementations currently; not tested for ANSI C/POSIX standard conformance.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
strand()	void srand(unsigned int *seed)	Seed a sequence of pseudorandom numbers for the rand() function. ANSI C standard.
random()	void random(unsigned int seed)	Initialize current pseudo-random number generation state array for the random() function based on seed. ANSI C standard.
sscanf()	sscanf(const char *str, const char *format, ...)	Read input from string. ANSI C standard.
toascii()	int toascii(int c)	Translate integer to a 7-bit ASCII character by zeroing higher-order bits. POSIX standard implementation (ANSI C extension function) in kernel/libc. Platform-specific macro implementations also exist in Cisco IOS software. For portability, use the standard libc version if possible.
va_arg()	type va_arg(va_list ap, type)	ANSI C standard macro to return next argument in varargs list. Conforming implementations available on different platforms at various release levels.
va_end()	void va_end(va_list ap)	ANSI C standard macro to invalidate a varargs list. Conforming implementations available on different platforms at various release levels.
va_start()	void va_start(va_list ap, argN)	ANSI C standard macro to initialize a varargs list. Conforming implementations available on different platforms at various release levels

File System and Stdio Functions (dirent.h, stdio.h, unistd.h)

access()	int access(const char *path, int amode)	Check accessibility of a file. POSIX standard, conformance to be checked.
chdir()	bool chdir(char *dir)	Change working directory. ANSI C standard.
closedir()	int closedir(DIR *dir)	Close directory stream. ANSI C standard.
close()	int close(int fd)	Close file descriptor. ANSI C standard.
fclose()	int fclose(FILE *stream)	Flush stream and close file descriptor. ANSI C standard.
fcntl()	int fcntl(int fd, int cmd, struct flock *lock)	Perform file control operations on an open file. ANSI C standard.
feof()	int feof(FILE *stream)	Test end-of-file indicator for a file stream. ANSI C standard.
fflush()	int fflush(FILE *stream)	Flush a file stream. ANSI C standard.
fgetc()	int fgetc(FILE *stream)	Get a byte from a file stream. ANSI C standard.
fgets()	char *fgets(char *s, int n, FILE *stream)	Get a string from a file stream. ANSI C standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
fileno()	int fileno(FILE *stream)	Get the file descriptor for a file stream. ANSI C standard.
fopen()	FILE *fopen(const char *filename, const char *mode)	Open a file stream. ANSI C standard.
fprintf()	int fprintf(FILE *stream, const char *fmt, ...)	Print formatted output to a file stream. ANSI C standard.
fputc()	int fputc(int c, FILE *stream)	Output a byte to a file stream. ANSI C standard.
fputs()	int fputs(const char *s, FILE *stream)	Output a string to a file stream. ANSI C standard.
fread()	size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream)	Read a number of items into a buffer (an array) from a file stream. ANSI C standard.
fscanf()	int fscanf(FILE *stream, const char *fmt, ...)	Read input from a file stream, interpreting and storing according to the provided format string. ANSI C standard.
fseek()	int fseek(FILE *stream, long offset, int whence)	Set the file position indicator in a file stream. ANSI C standard.
ftell()	long int ftell(FILE *stream)	Return the current file position indicator in a file stream. ANSI C standard.
fwrite()	size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream)	Write a number of items to a file stream. ANSI C standard.
getc()	int getc(FILE *stream)	Get a byte from a stream. Supported via internal _IO_getc macro that maps to getc().
ioctl()	int ioctl(int d, int request, ...)	Perform control operations on a STREAMS device. ANSI C standard.
mkdir()	int mkdir (const char* pathname, mode_t mode)	Create a new directory with the given pathname. ANSI C standard.
open()	int open(const char *pathname , int flags, mode_t mode)	Open a file. ANSI C standard.
opendir()	DIR *opendir(const char *name)	Open a directory stream. ANSI C standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
printf()	int printf(const char *fmt,...)	Print formatted output to a file stream. Cisco IOS printf() does not conform to ANSI C/POSIX standards and includes Cisco-specific format specifiers; see reference page for details.
posix_printf()	int posix_printf(const char *fmt,...)	<i>Available in 12.4T and 12.2S releases:</i> posix_printf() is a closely POSIX-compliant Cisco IOS version of printf(). Certain format specifiers are missing (%f, %e, E, %g, G because of a lack of floating-point support); these restrictions also apply to the other supported Cisco IOS posix_xxxprintf() functions in this table, which all share the same internal __posixdoprnt() function.
putc()	int putc(int c, FILE *stream)	Supported via internal _IO_putc macro that maps to putc().
read()	ssize_t read(int fd, void *buf, size_t count)	Read bytes from a file. ANSI C standard
readdir()	struct dirent *readdir(DIR *dir)	Read next directory entry in a directory stream. ANSI C standard.
readdir_r()	int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result)	readdir_r() is the thread-safe version.
remove()	int remove(const char *pathname)	Remove a file. ANSI C standard.
rename()	int rename(const char *old, const char *new)	Change the name of a file. ANSI C standard.
rewind()	void rewind(FILE *stream)	Reset the file position indicator in a file stream. ANSI C standard.
rmdir()	int rmdir(const char *path)	Remove a directory. ANSI C standard.
snprintf()	snprintf(char *s, const char *fmt,...)	Print formatted output up to a certain length to a string location. Various Cisco IOS implementations of the ANSI C standard extension function to sprintf(); for details, see the reference pages for each function.
cisco_snprintf()	cisco_snprintf(char *s, const char *fmt,...)	
posix_snprintf()	posix_snprintf(char *s, const char *fmt,...)	
snprintf_ansi_ret()	int snprintf_ansi_ret (char *cp, size_t size, const char *fmt,...)	POSIX-compliant version implemented as posix_snprintf() in some trains; snprintf_ansi_ret() mimics cisco_snprintf() but conforms to ANSI C standard for the return value.
sprintf()	sprintf(char *s, const char *fmt,...)	Cisco IOS implementation of the ANSI C standard; see the reference page for details.
posix_sprintf()	posix_sprintf(char *s, const char *fmt,...)	POSIX-compliant version implemented as posix_sprintf() in some trains.
stat()	int stat(const char *file_name, struct stat *buf)	Get file status and place it in a buffer. ANSI C standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
statvfs()	int statvfs(const char *path, struct statvfs *buf)	Get file system status information. ANSI C standard.
truncate()	int truncate(const char* path, off_t length)	Truncate a file to a specified length. ANSI C standard.
uname()	int uname(struct utsname *buf)	Get the current system name. ANSI C standard.
unlink()	int unlink(const char *path)	Remove a directory entry. ANSI C standard.
vfprintf()	int vfprintf(FILE *stream, const char *format, va_list ap)	Generates formatted output to a file stream. Implemented in Cisco IOS software as equivalent to Cisco IOS fprintf() using a varargs argument list.
vprintf()	int vprintf(const char *format, va_list ap)	Generates formatted output. Implemented in Cisco IOS software as equivalent to the Cisco IOS posix_printf() function, but using a varargs argument list.
vsnprintf()	vsnprintf(char *cp, size_t size, const char *fmt, va_list ap)	Cisco IOS implementation of the POSIX standard strings library extension function, equivalent to Cisco IOS sprintf() using a varargs argument list.
posix_vsnprintf()	posix_vsnprintf(char *s, size_t n, const char *format, va_list ap)	POSIX-compliant version implemented as posix_vsnprintf() in some trains.
vsprintf()	vsprintf(char *s, const char *fmt, va_list ap)	Cisco IOS implementation of the POSIX standard strings library extension function, equivalent to Cisco IOS sprintf() using a varargs argument list.
posix_vsprintf()	posix_vsprintf(char *s, const char *format, va_list ap)	POSIX-compliant version implemented as posix_vsprintf() in some trains.
write()	ssize_t write(int fd, const void *buf, size_t count)	Write bytes from a buffer to an open file. ANSI C standard.

String and Memory Copy Functions (string.h; cisco/include/cisolib.h <standards extension function declarations>; includes local functions)

memchr()	void *memchr(const void *src, int c, size_t length)	Find character in memory. ANSI C standard.
memcmp()	int memcmp(const void *s1, const void *s2, size_t n)	Compare two memory areas. ANSI C standard. memcmp() is an unsafe function. Use the memcmp_s() function instead.
memcpy()	void *memcpy(void *out, const void *in, size_t n)	Copy nonoverlapping memory areas. ANSI C standard. memcpy() is an unsafe function. Use the memcpy_s() function instead.
memmove()	oid *memmove(void *dst0, const void *src0, register size_t length)	Copy memory areas, handling possible overlap. More portable than bcopy() and memcpy(). Not required by ANSI C standard. memmove() is an unsafe function. Use the memmove_s() function instead.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
memset()	void *memset(const void *dst, int c, size_t length)	Fill memory with a value. ANSI C standard. memset() is an unsafe function. Use the memset_s() function instead.
strcasecmp()	strcasecmp (const char *s1, const char *s2)	Case-insensitive string comparison. Based on Berkeley Software Distribution (BSD) implementation, conforms to (extended) POSIX standards.
strcat()	char *strcat(char *destination, const char *source)	Concatenate strings. ANSI C standard. strcat() is an unsafe function. Use the strcat_s() function instead.
strchr()	char *strchr(const char *string, int c)	Search for character in string. ANSI C standard. strchr() is an unsafe function. Use the strfirstchar_s() function instead.
strcmp()	int strcmp(const char *a, const char *b)	Case-sensitive string match. ANSI C standard. strcmp() is an unsafe function. Use the strcmp_s() function instead.
strcoll()	int strcoll(const char *stra, const char *strb)	Locale-specific string match. ANSI C standard.
strcpy()	char *strcpy(char *destination, const char *source)	String copy without bounds-checking. ANSI C standard. strcpy() is an unsafe function. Use the strcpy_s() function instead. To avoid string overwrites for more secure code, use (nonstandard) Cisco IOS sstrncpy() instead. sstrncpy() is an unsafe function. Use the strncpy_s() function instead.
strcspn()	size_t strcspn(const char *string1, const char *string2)	Length of initial string segment that does <i>not</i> contain certain characters. ANSI C standard. strcspn() is an unsafe function. Use the strcspn_s() function instead.
strdup()	char *strdup(const char *s)	Duplicates a string. POSIX standard extension to ANSI C standard strings library.
strerror()	char *strerror(int errnum)	Converts error number to error string. Uses Cisco IOS COMP_INC(posix, errno.h) and COMP_INC(ifs, ifs_errno.h) as sources for string mappings. POSIX standard extension to ANSI C standard strings library.
strlcat()	size_t strlcat(char *dst, char const *src, size_t siz)	Appends characters of one string to another up to a certain size of the destination string (note the meaning of the <i>siz</i> differs from that of strncat()). Extension to ANSI C standard, conforms to industry implementations.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
strlcpy()	size_t strlcpy(char *dst, char const *src, size_t siz)	Copies at most a certain number of characters of a source string to the destination string. Extension to ANSI C standard, conforms to industry implementations.
strlen()	size_t strlen(const char *string)	Returns the number of nonNULL bytes in a string. ANSI C standard.
strncat()	char *strncat(char *destination, const char *source, size_t length)	Appends at most a certain number of characters of a source string to a destination string (note the <i>length</i> differs from that of <i>strlcat()</i>). ANSI C standard. <i>strncat()</i> is an unsafe function. Use the <i>strncat_s()</i> function instead.
strncmp()	int strncmp(const char *a, const char *b, size_t length)	Compare bytes of two strings. ANSI C standard. <i>strncmp()</i> is an unsafe function. Use the <i>strcmp_s()</i> function instead.
strncpy()	char *strncpy(char *destination, const char *source, size_t length)	Copy a certain number of bytes of a source string to a destination string. ANSI C standard. <i>strncpy()</i> is an unsafe function. Use the <i>strncpy_s()</i> function instead.
strnlen()	size_t strnlen(const char *str, size_t maxlen);	Returns length of a string, comparing at most a certain number of characters. ANSI C strings extension function in Cisco IOS software, not required by POSIX standards.
strupr()	char *strupr(const char *s1, const char *s2)	Find the first occurrence of specific characters in a string. ANSI C standard. <i>strupr()</i> is an unsafe function. Use the <i>strupr_s()</i> function instead.
strrchr()	char *strrchr(const char *string, int c)	Find the last occurrence of a specific character in a string. ANSI C standard. <i>strrchr()</i> is an unsafe function. Use the <i>strlastchar_s()</i> function instead.
strspn()	size_t strspn(const char *s1, const char *s2)	Get the length of a substring in a string consisting only of certain characters. ANSI C standard. <i>strspn()</i> is an unsafe function. Use the <i>strspn_s()</i> function instead.
strstr()	char *strstr(const char *s1, const char *s2)	Find the first occurrence of a certain substring in another string. ANSI C standard.
_strtok() _strtok_r()	char *strtok(char *source, const char *delimiters)	Break a string into a sequence of tokens starting with certain characters from another string. Also have reentrant-style (_r) implementation. ANSI C standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
<code>strtol()</code> <code>_strtol_r()</code>	<code>long strtol(const char *numptr, char **endptr, int base)</code>	Convert a string to its long integer equivalent. Also have reentrant-style (_r) implementation. ANSI C standard, supported in Cisco IOS via the internal <code>_strtol_internal()</code> function call.
<code>strtoul()</code>	<code>unsigned long strtoul(const char *numptr, char **endptr, int base)</code>	Convert a string to its unsigned long integer equivalent. ANSI C standard.
<code>tolower()</code>	<code>int tolower(int c)</code>	Replace uppercase character with its lowercase equivalent. ANSI C standard; previously implemented, conformance to be checked.
<code>toupper()</code>	<code>int toupper(int c)</code>	Replace lowercase character with its uppercase equivalent. ANSI C standard; previously implemented, conformance to be checked.
<code>wcstombs()</code>	<code>size_t wcstombs (char *s, const wchar_t *pwcs, size_t n)</code>	Convert wide character string to multibyte character string. Handling varies per locale. ANSI C standard.
<code>wctomb()</code>	<code>int wctomb(char *s, wchar_t wchar)</code>	Nonconforming minimal support implementation of ANSI C standard strings library function to convert wide character to multibyte character string.

Signal Functions (signal.h)

<code>alarm()</code>	<code>unsigned int alarm(unsigned int seconds)</code>	Schedule a <code>SIGALRM</code> signal. POSIX standard.
<code>sigprocmask()</code>	<code>int sigprocmask(int how, const sigset_t *set, sigset_t *oset)</code>	Examine or change the caller's signal mask (single-threaded process). POSIX standard.
<code>sigaddset()</code>	<code>int sigaddset(sigset_t *set, int signo)</code>	Add a signal number to a signal set. POSIX standard.
<code>sigdelset()</code>	<code>int sigdelset(sigset_t *set, int signo)</code>	Remove a signal number from a signal set. POSIX standard.
<code>sigismember()</code>	<code>int sigismember(const sigset_t *set, int signo)</code>	Test for a signal in a signal set. POSIX standard.
<code>signal()</code> Currently implemented in Cisco IOS software as: <code>posix_signal()</code>	<code>typedef void (*sighandler_t)(int sig) sighandler_t posix_signal(int signum, sighandler_t handler)</code>	Set up signal handling. Function currently implemented in Cisco IOS software that conforms to the POSIX standard is named <code>posix_signal()</code> in <code>cisco.comp/kernel/libc/src/signal/signal.c</code> as a workaround to accommodate issues with replacing existing platform-specific <code>signal()</code> (r4K) that is nonconforming.
<code>sigsuspend()</code>	<code>int sigsuspend(const sigset_t *sigmask)</code>	Set signal mask and suspend, awaiting a signal. POSIX standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
Semaphore Functions (semaphore.h, sys/sem.h)		
sem_close()	int sem_close(sem_t *sem)	Close a named semaphore. POSIX standard. Conforming implementation in 12.2S train. Currently stubbed in 12.4T train.
sem_destroy()	int sem_destroy(sem_t *sem)	Destroy an unnamed semaphore. POSIX standard. Conforming implementation in 12.2S train. Currently stubbed in 12.4T train.
sem_getvalue()	int sem_getvalue(sem_t *restrict sem, int *restrict sval)	Get the value of a semaphore. POSIX standard. Conforming implementation in 12.2S train. Currently stubbed in 12.4T train.
sem_init()	int sem_init(sem_t *sem, int pshared, unsigned value)	Initialize an unnamed semaphore to a certain value. POSIX standard. Conforming implementation in 12.2S-train. Currently stubbed in 12.4T train.
sem_open()	sem_t *sem_open(const char *name, int oflag, ...)	Initialize and open a named semaphore. POSIX standard. Conforming implementation in 12.2S-train. Currently stubbed in 12.4T train.
sem_post()	int sem_post(sem_t *sem)	Unlock a semaphore. POSIX standard. Conforming implementation in 12.2S-train. Currently stubbed in 12.4T train.
sem_trywait()	int sem_trywait(sem_t *sem)	Lock a semaphore if currently not locked. POSIX standard. Conforming implementation in 12.2S train. Currently stubbed in 12.4T train.
sem_unlink()	int sem_unlink(const char *name)	Remove a named semaphore. POSIX standard. Conforming implementation in 12.2S train. Currently stubbed in 12.4T train.
sem_wait()	int sem_wait(sem_t *sem)	Lock a semaphore, waiting if necessary until it can be locked or a signal occurs. POSIX standard. Conforming implementation in 12.2S train. Currently stubbed in 12.4T train.
semctl()	int semctl(int semid, int semnum, int cmd, ...)	Semaphore control on X/Open System Interface Extension (XSI) semaphores. POSIX standard. Only binary counting is supported. Examples: semctl(lckID,0,IPC_RMID,0) semctl(lckID,1,SETVAL,semctl_arg) semctl(lckID,1,GETVAL,0)
semget()	int semget(key_t key, int nsems, int semflag)	Get a set of XSI semaphores. POSIX standard.
semop()	int semop(int semid, struct sembuf *sops, unsigned nsops)	Perform semaphore operations on a set of XSI semaphores. POSIX standard.

Socket Functions (sys/socket.h): Implemented in Release 12.2S/12.4T as calls to Cisco IOS socket_xxx_equivalent functions with limited POSIX standard conformance

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
accept()	int accept(int <i>socket</i> , struct sockaddr * <i>address</i> , socklen_t * <i>address_len</i>)	Accept new connection on a socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_accept()</i> function (SCTP Sockets).
bind()	int bind(int <i>socket</i> , const struct sockaddr * <i>address</i> , socklen_t <i>address_len</i>)	Bind a socket address (name) to a socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_bind()</i> function (SCTP Sockets).
connect()	int connect(int <i>socket</i> , const struct sockaddr * <i>address</i> , socklen_t <i>address_len</i>)	Connect a socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_connect()</i> function (SCTP Sockets).
FD_CLR FD_ISSET FD_SET FD_SETSIZE FD_ZERO	void FD_CLR(int <i>fd</i> , fd_set * <i>fdset</i>) int FD_ISSET(int <i>fd</i> , fd_set * <i>fdset</i>) void FD_SET(int <i>fd</i> , fd_set * <i>fdset</i>) void FD_ZERO(fd_set * <i>fdset</i>)	Macros to initialize, set, clear, and test file descriptor masks of type <i>fd_set</i> , as related to the <i>select()</i> function. In 12.2S and 12.4T trains: POSIX-compliant macros are in the componentized include file COMP_INC(posix, sys/select.h).
getpeername()	int getpeername(int <i>socket</i> , struct sockaddr * <i>address</i> , socklen_t * <i>address_len</i>)	Retrieve the peer address of a socket. In 12.2S and 12.4T trains: Implemented by a call to the Cisco IOS <i>socket_get_peername()</i> function (SCTP Sockets).
getsockname()	int getsockname(int <i>socket</i> , struct sockaddr * <i>address</i> , socklen_t * <i>address_len</i>)	Retrieve the locally bound socket address (name) of a socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_get_localname()</i> function (SCTP Sockets).
getsockopt()	int getsockopt(int <i>socket</i> , int <i>level</i> , int <i>option_name</i> , void * <i>option_value</i> , socklen_t * <i>option_len</i>)	Retrieve socket options. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_get_option()</i> function (SCTP Sockets).
listen()	int listen(int <i>socket</i> , int <i>backlog</i>)	Mark a connection mode socket as accepting connections, and limit the queue of incoming connections. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_listen()</i> function (SCTP Sockets).
recv()	ssize_t recv(int <i>socket</i> , void * <i>buffer</i> , size_t <i>length</i> , int <i>flags</i>)	Receive a message from a connected socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_recv()</i> function (SCTP Sockets).

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
recvfrom()	ssize_t recvfrom(int <i>socket</i> , void * <i>buffer</i> , size_t <i>length</i> , int <i>flags</i> , struct sockaddr * <i>address</i> , socklen_t * <i>address_len</i>)	Receive a message from a connection or connectionless mode socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_recvfrom()</i> function (SCTP Sockets).
recvmsg()	ssize_t recvmsg(int <i>socket</i> , struct msghdr * <i>message</i> , int <i>flags</i>)	Receive a message from an unconnected or connected socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_recvmsg()</i> function (SCTP Sockets).
select()	int select(int <i>nfds</i> , fd_set * <i>readfds</i> , fd_set * <i>writefd</i> s, fd_set * <i>errorfd</i> s, struct timeval * <i>timeout</i>)	Synchronously test file descriptors ready for processing (reading, writing, other conditions). In 12.2S train <i>only</i> : Implemented by call to Cisco IOS <i>socket_select()</i> function (SCTP Sockets).
send()	ssize_t send(int <i>socket</i> , const void * <i>buffer</i> , size_t <i>length</i> , int <i>flags</i>)	Initiate transmission of a message on a socket to its peer. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_send()</i> function (SCTP Sockets).
sendmsg()	ssize_t sendmsg(int <i>socket</i> , const struct msghdr * <i>message</i> , int <i>flags</i>)	Send a message on a socket using a message structure. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_sendmsg()</i> function (SCTP Sockets).
sendto()	ssize_t sendto(int <i>socket</i> , const void * <i>message</i> , size_t <i>length</i> , int <i>flags</i> , const struct sockaddr * <i>dest_addr</i> , socklen_t <i>dest_len</i>)	Send a message on a connection mode or connectionless mode socket. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_sendto()</i> function (SCTP Sockets).
setsockopt()	int setsockopt(int <i>socket</i> , int <i>level</i> , int <i>option_name</i> , const void * <i>option_value</i> , socklen_t <i>option_len</i>)	Set socket options. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_set_option()</i> function (SCTP Sockets).
shutdown()	int shutdown(int <i>socket</i> , int <i>how</i>)	Shut down socket send and receive operations. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_shutdown()</i> function (SCTP Sockets).
socket()	int socket(int <i>domain</i> , int <i>type</i> , int <i>protocol</i>)	Create an unbound socket (endpoint for communication) in a communications domain. In 12.2S and 12.4T trains: Implemented by a call to Cisco IOS <i>socket_open()</i> function (SCTP Sockets), which performs the equivalent of the standard library <i>socket()</i> and <i>open()</i> functions.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
socketpair()	int socketpair(int <i>domain</i> , int <i>type</i> , int <i>protocol</i> , int <i>socket_vector</i> [2])	Create an unbound pair of connected sockets (endpoints for communication) in a communications domain of a specified type (with optionally specified protocol). In 12.2S train <i>only</i> : POSIX standard socket function to set up a connected socket pair, implemented in Cisco IOS software by calling <code>socket()</code> , which maps to Cisco IOS <code>socket_open()</code> (SCTP Sockets).
Time Functions (time.h, sys/time.h, sys/times.h)		
asctime()	char *asctime(const struct tm * <i>timeptr</i>)	Convert date and time to a string. POSIX standard.
asctime_r()	char *asctime_r(const struct tm *restrict <i>tm</i> , char *restrict <i>buf</i>)	Thread-safe version of <code>asctime()</code> . POSIX standard.
clock()	clock_t clock(void)	Return best approximation of processor time used by a process over a particular era related to the process invocation. POSIX standard.
clock_getres()	int clock_getres(clockid_t <i>clock_id</i> , struct timespec * <i>res</i>)	Get the resolution of a clock. POSIX standard.
clock_gettime()	int clock_gettime(clockid_t <i>clock_id</i> , struct timespec * <i>tp</i>)	Get the current time setting of a clock. POSIX standard.
clock_settime()	int clock_settime(clockid_t <i>clock_id</i> , const struct timespec * <i>tp</i>)	Set the time of a clock to a certain value. POSIX standard.
ctime()	char *ctime(const time_t * <i>ltime</i>)	Convert a time value to a date and time string. POSIX standard.
ctime_r()	int clock_nanosleep(clockid_t <i>clock_id</i> , int <i>flags</i> , const struct timespec * <i>rqtp</i> , struct timespec * <i>rmtip</i>)	Thread-safe version of <code>ctime()</code> . POSIX standard.
difftime()	time_t difftime(time_t <i>time1</i> , time_t <i>time0</i>)	Return the difference between two calendar time values. POSIX standard.
getdate()	struct tm *getdate(const char * <i>string</i>)	Convert string representation of date and time to a structure. POSIX standard.
gettimeofday()	int gettimeofday(struct timeval * <i>tv</i> , struct timezone * <i>tz</i>)	Get current time-of-day and represent in a POSIX <code>timeval</code> structure. POSIX standard.
gmtime()	struct tm* gmtime(const time_t * <i>timer</i>)	Convert time value in seconds into a structure expressed as Coordinated Universal Time (UTC). POSIX standard.
gmtime_r()	struct tm *gmtime_r(const time_t *restrict <i>timer</i> , struct tm *restrict <i>result</i>)	Thread-safe version of <code>gmtime()</code> . POSIX standard.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
localtime()	struct tm* localtime(const time_t *ltime)	Convert time value in seconds into a structure expressed as a local time, correcting for the local time zone and seasonal time adjustments if necessary.
localtime_r()	struct tm *localtime_r(const time_t *restrict timer, struct tm *restrict result)	Thread-safe version. POSIX standard.
mktime()	time_t mktime(struct tm *timeptr)	Convert a time represented as a time structure into a time value in seconds since the reference epoch. POSIX standard.
strftime()	size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timptra)	Convert date and time to a string. Conforms closely to POSIX standard, except not all format specifiers are supported: uses format_time() function; see that reference page for supported formats.
strptime()	char *strptime(const char *restrict buf, const char *restrict format, struct tm *restrict tm)	Convert data and time to a string using a particular format. POSIX standard.
time()	time_t time(time_t *tloc)	Get the current time in seconds since the reference epoch. POSIX standard.
times()	clock_t times(struct tms *buf)	Get time-accounting information for the calling process in a tms structure (defined in sys/times.h). POSIX standard.
tzset()	void tzset(void)	Use TZ environment variable to set time zone information required by other time functions. POSIX standard.

Timer Functions (time.h, sys/time.h)

getitimer()	int getitimer(int which, struct itimerval *value)	Get the value stored in the specified interval timer. Conforms to POSIX standard. Implemented in 12.2S train only.
setitimer()	int setitimer(int which, const struct itimerval *restrict value, struct itimerval *restrict ovalue)	Set the value of the specified interval timer. Conforms to POSIX standard. Implemented in 12.2S train only.
timer_create()	int timer_create(clockid_t clockid, struct sigevent *restrict evp, timer_t *restrict timerid)	Create a per-process timer. Conforms to POSIX standard. Implemented in 12.2S train only.
timer_delete()	int timer_delete(timer_t timerid)	Delete a per-process timer. Conforms to POSIX standard. Implemented in 12.2S train only.
timer_settime()	int timer_settime(timer_t timerid, int flags, const struct itimerspec *restrict value, struct itimerspec *restrict ovalue)	Set the expiration time of the specified timer. Conforms to POSIX standard. Implemented in 12.2S train only.

Table 11-3 Available ANSI C, POSIX, and Related Library Functions (continued)

Function Name	Function Prototype	Description, Compliance, and Other Notes
timer_gettime()	int timer_gettime(timer_t timerid, struct itimerspec *value)	Get the specified timer expiration time and store in the specified structure. Conforms to POSIX standard. Implemented in 12.2S-train only.

Note All the other functions documented in this chapter are the Cisco-specific or other versions of ANSI C standard functions; however, the Cisco-specific or other versions often are *not* required in the ANSI C or POSIX standards and do *not* require any supporting operating system subroutines. See the “Portability” section in each function’s reference page for more information.)

For more recently ported functions, we do not duplicate the standards documentation for functions that conform to the POSIX standards and have no notable API differences. At this time, for reference pages on POSIX-compliant functions in the Cisco IOS software that are not documented in the *Cisco IOS API Reference*, see section 11.4.2.211.4.2.2, “Accessing POSIX Standards Reference Documentation,” for details about online access to the POSIX standards reference documentation.

11.4.2.1 POSIX Standard Header Files in Cisco IOS Source Code

In release trains 12.2S, 12.4T, and later in which POSIX functions have been implemented:

In Cisco IOS software, function prototypes for POSIX-compliant standard library functions are in the appropriate include files in the POSIX standard library function source tree. Function declarations should be included using the `COMP_INC` definition to automatically use the proper include file with operating system-specific mappings:

```
COMP_INC(posix, ...)
```

11.4.2.2 Accessing POSIX Standards Reference Documentation

The Open Group, a neutral consortium in the standard’s development and maintenance group, provides printed and Web-accessible documentation of the standards’ definitions and system interfaces as part of a multiple-volume documentation set, The Single UNIX Specification, Version 3 (includes IEEE POSIX, The Open Group’s Technical Standard, and ISO/IEC designations). This documentation set is available via the following website:

<http://www.opengroup.org>

You can access the POSIX standards reference documentation on which the Cisco IOS standard API functions are based through The Open Group’s catalog website:

<http://www.opengroup.org/bookstore/catalog>

The following documents are available to view online or download for local, personal use at no charge:

- System Interfaces Volume (API details):

<http://www.opengroup.org/bookstore/catalog/c047.htm>

- Base Definitions Volume (architecture and header file details):

<http://www.opengroup.org/bookstore/catalog/c046.htm>

The steps to access POSIX standard API reference pages from The Open Group site are provided next.

For Web-based access to the POSIX standard library API Reference pages:

- Step 1** Select the **FREE HTML** view/download option in the lower right side of the display on either of the above-mentioned pages to access the index page for both documents. See Figure 11-1. You might be prompted with a registration screen before being allowed access for the first time only; enter your e-mail address to register, then proceed to view the documents.

Figure 11-1 The Open Group POSIX Reference (HTML View/Download Option)

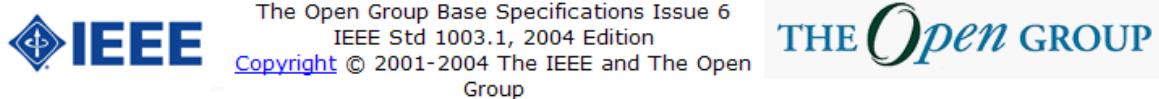
System Interfaces, Issue 6, 2004 Edition [XSH]	Availability
<p>This document is a revision of the System Interfaces volume of IEEE Std 1003.1-2001 and ISO/IEC 9945:2003, incorporating IEEE Std 1003.1-2001/Cor 1-2002 (identical to ISO/IEC 9945:2002/Cor 1:2003) and IEEE Std 1003.1-2001/Cor 2-2004 (identical to ISO/IEC 9945:2003/Cor 1:2004). This document forms part of the Single UNIX Specification, Version 3 (T041). Definitions in common with Shell and Utilities, Issue 6 can be found in Base Definitions, Issue 6. This document includes extensive revisions to the Issue 5 specification so that it is an IEEE Standard, and ISO/IEC Standard, and an Open Group Technical Standard. These include: networking interfaces incorporated from XNS, Issue 5.2; the addition of new functionality for alignment with ISO/IEC 9899:1999 (ISO C); updates to reflect changes in the IEEE P1003.1a draft standard; incorporation of IEEE Std 1003.1d-1999 and IEEE Std 1003.1j-2000; incorporation of IEEE PASC Interpretations and The Open Group</p>	<p>Electronic Publication Only (hard copy not available)</p> <p>Price US\$ 60.00 (payment also accepted in GBP/EUR at checkout)</p> <p>Shipping Free or Not Applicable</p> <p> Buy now by secure credit/debit card transaction</p> <p>PDF file available for immediate download after purchase</p>

FREE HTML
See the HTML version on the web

FREE PDF
PDF version free only to members, subject to The Open Group terms :

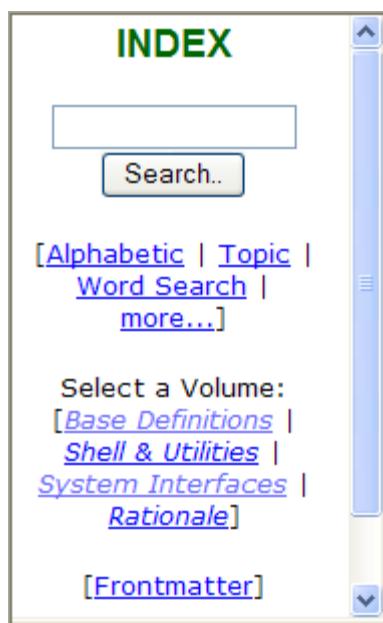
- Step 2** After access is established the first time, you can bookmark the site and access it directly in the future. The reference page display panel has the heading shown in Figure 11-2.

Figure 11-2 The Open Group POSIX Reference Documentation Portal Heading



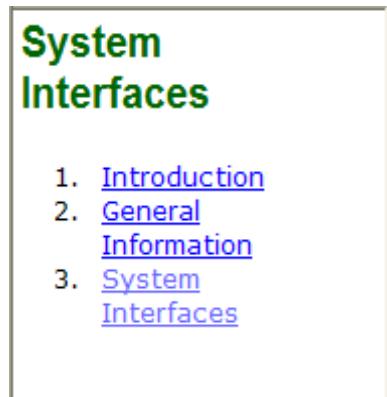
- Step 3** Select the **System Interfaces** Volume from the **INDEX** navigation window on the upper left-hand side of the page, as shown in Figure 11-3.

Figure 11-3 System Interfaces Volume Selection from INDEX Navigation Window



- (a) Select the **3. System Interfaces** section in the lower left-hand navigation pane, as shown in Figure 11-4, for an alphabetical index into all API pages in the specification.

Figure 11-4 Accessing the System Interfaces API Reference Index



- (b) Scroll down the list of API functions (see Figure 11-5) in the lower left-hand navigation pane, and click on a function name to display its reference page in the display pane on the right. Note that only some of the functions and macros in this navigation pane are supported in Cisco IOS software. Only the functions listed in the *Cisco IOS Programmer's Guide*, Table 11-3, are currently available.

Figure 11-5 Alphabetical List of POSIX Standard API Reference Pages



11.5 Absence of Floating Point Functions

Cisco IOS software reserves floating point registers in a nonstandard manner, and hence the standard floating point (`math.h`) functions are not allowed for the following reasons:

Absence of Floating Point Functions

- Floating point functions are not portable across all platforms because some platforms on which Cisco IOS software runs do not have an FPU (A floating point unit, usually in a single integrated circuit, possibly on the same IC as the central processing unit).
- Cisco IOS interrupt and process swap routines stash values in floating point registers for efficiency. This is non-ABI compliant use of these registers. Therefore, if you got interrupted during any floating point operation, data corruption or crash would result with Cisco IOS software.
- Use of the floating point libraries could affect performance in Cisco IOS software.

The workaround to this Cisco IOS restriction is to emulate FP (Floating Point) using fixed point and the integer unit. For example:

- See `cm_docsis_equalizer.c` on the `v122_15_cz_throttle` branch for an example where fixed point math using the integer unit was used in place of using FP libraries.
- See `log10_t10K`, an internally generated log function.

Note Because of this limitation on floating point function support, Cisco IOS output and string formatting functions (such as `printf()` and related functions) generally do not support standard floating point format specifiers.

CNS

12.1 Introduction

This chapter discusses two parts of CNS.

- Event Agent Services
- Config Agent Services

The chapter assumes knowledge of the basic terms and concepts of CNS technology. It focuses on how to use the CNS API. If you need to become familiar with CNS basics, please consult one or more of the documents listed in the “Related Reading” section at the end of this chapter.

Note Cisco IOS CNS development questions can be directed to the cns-ios-sw@cisco.com alias.

12.1.1 Terms

CCITT

Consultative Committee for International Telegraph and Telephone. International organization responsible for the development of communications standards. Now called the ITU-T. See also ITU-T.

DEN

Directory Enabled Networking. DEN specification describes an information model of network elements, protocols, services, and their relationships. A key foundation for policy-based networking and intelligent network services, DEN can fundamentally change how interoperable network applications are developed and used.

ISV

An ISV (independent software vendor) makes and sells software products that run on one or more computer hardware or operating system platforms.

LDAP

Lightweight Directory Access Protocol. Protocol that provides access for management and browser applications that provide read/write interactive access to the X.500 Directory.

ITU-T

International Telecommunication Union Telecommunication Standardization Sector. International body that develops worldwide standards for telecommunications technologies. The ITU-T carries out the functions of the former CCITT. See also CCITT.

X.500

ITU-T recommendation specifying a standard for distributed maintenance of files and directories.

12.1.2 CNS Overview

Cisco Networking Services (CNS) is a set of Directory Enabled Networking (DEN) software development tools, including an implementation of DEN and DEN-derived Cisco LDAP schema, LDAP client software, and scalable event services. CNS is a controlled release, available to selected network ISV, SI, and service provider partners, as well as internal Cisco engineering teams. End-user customers will benefit from CNS via directory-enabled products that Cisco and Cisco partners bring to the market. The Cisco Networking Services (CNS) projects are delivering next-generation technologies to drive more intelligence into the network and reduce the need for human intervention.

12.1.3 Event Agent Services

Event Agent Services provide the infrastructure for Cisco IOS applications to exchange data with peer applications at remote network locations. Event Agent services present applications with an event bus network architecture featuring event-driven messaging. An *event* consists of a text subject header and its associated binary payload. Cisco IOS applications assume the role of producer or consumer, a producer being the sender of events while a consumer is the receiver of events. An application can be both a producer and a consumer. There is no need for the application to locate peer applications or to determine their network addresses. The network details required to exchange events between two applications are encapsulated within the Event Agent, hidden from the applications. An application sets itself up to receive events by *subscribing* to the appropriate event *subject name*. It sends events by *producing* to the appropriate *subject name*.

Subject names only need to be unique within Cisco IOS. There is a name space mapper that can be configured to provide network unique subject names. Usually, this is done by appending the device-id to the subject. Be sure to use the Namespace Mapper (NSM). See ENG-78026 for information on NSM. Also, the NSM server is described in EDCS-148500 and the NSM client API library is discussed in EDCS-148499.

12.1.3.1 Starting an Event Agent Session

Before an application can use one of the Event Agent services it must initiate an Event Agent session by making a `cns_ea_open()` call.

```
ea_handle cns_ea_open(watched_queue *queue, ulong flags);
```

The `cns_ea_open()` call returns a session handle to the application. This session handle must be passed to the Event Agent in all subsequent Event Agent library API calls. An application can only be engaged in a single Event Agent session. A second `cns_ea_open()` without a prior `cns_ea_close()` is rejected.

All the Event Agent API functions return `EA_SUCCESS` or `EA_FAIL`. To get more information about an `EA_FAIL` condition, an application may look into the `ea_err_info` structure, which is defined in `cns_event_agent_api.h`. For example, in case of `EA_FAIL` of `cns_ea_open()`, the application may check detailed information by reading

```
ea_err_info.on_api.ea_open.reason.
```

12.1.3.2 Sending an Event

An application calls the `cns_ea_produce()` function to send an event with the specified subject and data payload to the event bus.

```
int cns_ea_produce(ea_handle handle,
                    char* subject_name,
                    int data_length,
                    char* data);
```

12.1.3.2.1 Example

The following example shows the `cns_ea_produce()` function being used to send an event with `SUBJ_CONFIG_ID_CHANGED`. It can be found in `/vob/ios/sys/cns/config/cns_config.c`.

```
/*
 * Function Name: cns_output_config_id_changed_msg
 *
 * Description:
 *     This routine outputs a config id changed message to
 *     subject: config-id-changed.
 *
 * Inputs:
 *     time          time of config change
 *     new_cfg_id    new config id
 *     old_cfg_id    old config id
 *
 * Outputs:
 *
 * Return:
 *     NO_ERROR:           successful.
 *     NULL_POINTER_ERROR: invalid connection handle.
 *     XML_ENCODE_ERROR:   error when encoding the XML warning message
 *
 */
int cns_output_config_id_changed_msg (char *time, char *new_cfg_id,
                                      char *old_cfg_id)
{
    /*
     * write to buf and send event
     */
    char *workbuf;
    unsigned int length = 0;
    int rc = 0;

    /*
     * make sure that the queue handle for posting to the 'event agent'
     * is available
     */
    if (FALSE == get_and_set_ea_open_handle() ) {
        return (NO_EVENT_AGENT_HANDLE);
    }

    /*
     * call xml encode twice:
     * -first time: pass NULL buffer pointer to calculate length only
     * -2nd    time: pass valid buffer pointer to move data into the buffer
     */
    xml_encode_cfg_id_changed(CONFIG_ID_CHANGED, time, new_cfg_id,
```

```

        old_cfg_id, NULL, &length);

workbuf = malloc(length + 1);
if (workbuf == NULL) {
    return (MALLOC_ERROR);
}
xml_encode_cfg_id_changed(CONFIG_ID_CHANGED, time, new_cfg_id,
    old_cfg_id, workbuf, &length);

cns_print_debug_message(DEBUG_MASK_CNS_CFG_AGENT,
    ("\n%s(): config id changed = %s", __FUNCTION__, workbuf));

if (CNS_SUCCESS != (rc = ea_produce(cns_get_cfg_agent_event_handle(),
    SUBJ_CONFIG_ID_CHANGED, length, workbuf))) {
    cns_print_debug_message(DEBUG_MASK_CNS_CFG_AGENT,
        ("\n%s(): Error sending event", __FUNCTION__));
    return rc;
}

free(workbuf);
return (NO_ERROR);

} /* end of function: cns_output_config_id_changed_msg */

```

12.1.3.3 Receiving an Event

Conversely, an application calls the [cns_ea_subscribe\(\)](#) function to receive events from the event bus.

```
int cns_ea_subscribe(ea_handle handle, char* subject_name);
```

Subscribing to a subject name tells the Event Agent that the subscriber is receiving all the events sent by all event bus publishers with a subject matching the one specified in the [cns_ea_subscribe\(\)](#) call. An application may make multiple [cns_ea_subscribe\(\)](#) calls to subscribe to multiple subjects. The application can unsubscribe from a subject at any time.

Events are delivered to the application asynchronously in a queue. Applications may choose for the Event Agent to manage the queue in which case the Event Agent creates an Cisco IOS watched queue for the application and enqueues messages in the queue as they are received.

12.1.3.4 Waiting for Messages

Once an application has subscribed to all the subjects it is interested in, it has to wait for messages. To accomplish that, the application needs to call [cns_ea_read\(\)](#).

```
int cns_ea_read(ea_handle handle, pea_msg p_ea_msg);
```

The [cns_ea_read\(\)](#) blocks until an event is received from the event bus or an error occurs. The [cns_ea_read\(\)](#) returns after receiving each event.

12.1.3.5 Registering a Callback Function

Optionally, an application may register a callback function by calling [cns_ea_set_callback\(\)](#) and waiting for messages by calling [cns_ea_read\(\)](#).

```
int cns_ea_set_callback (ea_handle handle, event_read_f callback);
```

When a callback function is specified, the Event Agent invokes the callback function once for each subscribed event received.

12.1.3.6 Setting the Context

The application may also set the context using `cns_ea_set_context()`, which the Event Agent passes unchanged to the callback function. To store the context value that is passed to the callback function during `cns_ea_read()`, call the `cns_ea_set_context()` function.

```
int cns_ea_set_context (ea_handle handle, void *context);
```

An application may have more than one watched queue. If the application unblocks for some reason other than a subscribed event, `cns_ea_read()` will return `EA_FAIL`.
`ea_err_info.on_api.ea_read.reason` will be set to `EA_EINTR` with the major and minor code of the `process_get_wakeup()` call stored in `ea_err_info.on_api.ea_read.major` and `minor`.

Applications are sent the status message, `STATUS_CODE_EA_CLOSED`, if the Event Agent is unconfigured.

12.1.3.7 Storing the Callback Function Pointer

When the Event Agent is restarted, a callback function pointer is invoked. To store this callback function pointer, call the `cns_ea_set_restart()` function.

```
int cns_ea_set_restart (event_restart_f callback, void *context);
```

12.1.3.8 Clearing the Restart Callback Notification

To clear the restart callback notification when it terminates, call the `cns_ea_clear_restart()` function.

```
int cns_ea_clear_restart (event_restart_f callback, void *context);
```

If an application uses `cns_ea_set_restart()` for notification of the Event Agent restarting, it should also call `cns_ea_clear_restart()`, when the application itself is terminated.

12.1.3.9 Setting Up a Watched Boolean

To set up a watched Boolean for the case when **no cns event** happens, call the `cns_ea_setup()` function.

```
void cns_ea_setup(char *process_name, watched_boolean **my_restart);
```

`cns_ea_setup()` and related functions can be used even if an application manages its own queue.

12.1.3.10 Reading Received Event Subjects and Payloads

To read received event subjects and payloads from application queue message units, call the `cns_ea_extract_msg()` function.

```
int cns_ea_extract_msg (pea_msg p_ea_msg, void *p_queue_item);
```

An application may choose to manage its own queue. In that case, it needs to create a queue and pass the queue pointer as an argument in the `cns_ea_open()` call. The Event Agent enqueues opaque event items to the specified queue in sequence as they arrive and it is the application's responsibility

to dequeue them. The application needs to dequeue each opaque event item from the queue and then call `cns_ea_extract_msg()` to extract the application-readable event subject and payload from the opaque event item.

12.1.3.11 Stopping Receiving Events

To stop receiving events for a specified subject, call the `cns_ea_unsubscribe()` function.

```
int cns_ea_unsubscribe(ea_handle handle, char* subject_name);
```

12.1.3.12 Freeing Storage

The application must issue a `cns_ea_free()` function call to free a message's subject and data fields if they were allocated by the Event Agent.

```
void cns_ea_free(pea_msg ea_message);
```

If an application supplies its own data and subject space, then it has to manage it.

12.1.3.13 Stopping an Event Agent Service Session

When an application is done with the Event Agent Service, it calls the `cns_ea_close()` function call to terminate the session.

```
int cns_ea_close(ea_handle handle);
```

To use the Event Agent again, a new session must be started with the `cns_ea_open()` function.

12.1.3.14 Notification When the Event Agent Shuts Down

Should the Event Agent shut down, which is a highly unlikely occurrence, it calls the `STATUS_CODE_EA_CLOSED` status message to notify all applications. Applications dequeuing regular messages should check for status messages, which are defined by `EA_STATUS` in the `msg_type` field of the `ea_msg` structure defined in `event_agent_api.h`.

12.1.4 Considerations for Using the Event Agent Service

The following commands must be issued as discussed to prevent memory leaks, save CPU cycles, and use the Event Agent Services efficiently.

12.1.4.1 Calling ea_free() for Each ea_read()

`cns_ea_free()` frees the subject or data storage allocated by the `cns_ea_read()` function. The `cns_ea_free()` function does not free the subject or data storage that is provided by the application.

12.1.4.2 Unsubscribing to All Subjects No Longer Required

Subscribing to subject names requires storage and CPU cycle overhead even when no messages are arriving. Unsubscribing to subjects that are no longer needed frees up subject name entry storage and speeds up the search routine.

12.1.4.3 Calling ea_close() to Terminate the Event Agent Services Gracefully

Use `cns_ea_close()` to terminate the Event Agent Services when done with the Event Agent. The `cns_ea_close()` performs house cleaning functions that free up storage and reduce CPU cycles.

12.1.4.4 Not Supporting Wild Card Subject Names

Wild card characters are not allowed in any part of the subject name.

12.1.5 Config Agent Services

Config Agent services provide the infrastructure for Cisco IOS configuration to be applied remotely. In addition Cisco IOS applications have access to several key informational and configuration aspects to this agent. The router must identify itself to the config server, typically a non-Cisco IOS application running on an external appliance. This identification must be unique to the server's name space. The functions presented here provide a mechanism to produce a unique identifier by whatever means the interested Cisco IOS application needs to use.

12.1.5.1 Registering a Callback Function Invoked by the Config Agent

To register a callback function invoked by the Config Agent as an invitation to override the Config Agent's default assignment of `config_id`, call the `cns_config_id_mode_reg()` function.

```
int cns_config_id_mode_reg (cns_id_mode_set_f func, void *context);
```

12.1.5.2 Unregistering a Callback Function Invoked by the Config Agent

To unregister a callback function invoked by the Config Agent as an invitation to override the Config Agent's default assignment of `config_id`, call the `cns_config_id_mode_unreg()` function.

```
void cns_config_id_mode_unreg (cns_id_mode_set_f func);
```

12.1.5.3 Commanding the Config Agent to Assign the config_id to the Value of the Argument String

To command the Config Agent to assign the `config_id` to the value of the argument string, call the `cns_config_id_set()` function.

```
int cns_config_id_set (const char *config_id);
```

12.1.6 Related Reading

There are several concise resources including:

- [1] 'CNS Schema Phase 1 Program Plan', Alex Wang, ENG-48061- November 1999.
- [2] 'IOS CNS Event Service Client Software Unit Design Specification', Fan Jiao, ENG-26208 - March 1999
- [3] 'CNS IOS Configuration Agent 1.0 Software Unit Design Specification', ENG-59810 - May 2000.
- [4] 'CNS Schema V1.0 System Functional Specification', John Strassner & Alex Wang, ENG-48421 - November 1999.
- [5] 'EBMU Event Services Software (ESS) Primer and Overview', ENG-110601 - March 2001.

Introduction

[6] 'CNS IOS Event Agent System Functional Specification', James Chuang, ENG-46698 - September 2000.

[7] 'Event Agent Software Unit Design Specification', Hugh Wong, ENG-58266 - September 2000.

P A R T 3

Kernel Support Services

Subsystems

This chapter describes the programming interface and developer hooks for defining subsystems.

13.1 Overview: Subsystems

In general, a subsystem is a discrete code module that supports various functions of an embedded system. Segregating IOS into subsystems allows images to be compiled with a minimum of link requirements.

There are two usages of the term “subsystem” in IOS:

- 1 A runtime subsystem, an independent entry point into an IOS application. This is the most common intention of the term subsystem in IOS. A runtime subsystem is defined within a .c file using the `SUBSYS_HEADER()` macro.
- 2 A compile-time subsystem, which defines the modules to be included together during system compilation. A compile-time subsystem is defined within a makefile using a set of build instructions of the form `sub_xxx`.

Conceptually, these two different subsystem types are quite separate. However, the most typical implementation is a one-to-one relationship between a runtime subsystem and a compile-time subsystem. A one-to-one example is the `sub_atm_common` compile-time subsystem because the ATM parser code contains a single runtime subsystem.

It is not unusual for a single compile-time subsystem to have several runtime subsystems (as defined by the `SUBSYS_HEADER()` macro, discussed later in this chapter). An example of multiple runtime subsystems in a single compile-time subsystem is the `sub_arp` compile-time subsystem because the Address Resolution Protocol has a `REGISTRY` subsystem and another portion that is a `KERNEL` subsystem.

You can also define `sub_xxx` modules containing just a set of platform-specific functions that do not need a `SUBSYS` header.

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.

Note Cisco IOS subsystems questions can be directed to the `interest-os@cisco.com` alias.

13.2 Runtime Subsystems

Runtime subsystems refer to the more common usage of the term “subsystem”, which are independent code sections that run during system initialization.

13.2.1 Defining a Runtime Subsystem

These independent code sections are each identified to the IOS system by coding a SUBSYS_HEADER() macro shown here:

```
SUBSYS_HEADER(char *name,
              ulong major_version, ulong minor_version, ulong edit_version,
              void *init, ulong class, char *property);
```

These arguments are all discussed in Chapter 13, “[Subsystems](#)”, of the *Cisco IOS API Reference*.

13.2.2 Subsystem Name

Each runtime subsystem has a name, specified as the first argument in the SUBSYS_HEADER() macro:

```
SUBSYS_HEADER(char *name, . . .);
```

This name is used by the intra-class sequencing property described in subsection 13.2.6, “Intra-Class Sequencing Property”.

13.2.3 Subsystem Entry Point

Each runtime subsystem has a single entry point, the “init” routine, which starts the subsystem during system initialization. The “init” routine is specified in the SUBSYS_HEADER() macro:

```
SUBSYS_HEADER(. . ., void *init, . . .);
```

The convention for the initialization function name is *<subsys_name>_init()*.

The purpose of the entry point is to complete any initialization processing required for the subsystem to begin to function appropriately in the system, such as creating new processes to be scheduled for execution and registering for future callbacks. It is invoked only once at subsystem start time.

The Cisco IOS subsystem initialization code searches through the data segment of the system image for subsystem headers, identified by a subsystem header *magic number* defined in header file “*.../subsys.h*”. It creates a list of subsystem headers for subsystems to initialize, and based on information in the subsystem headers, determines the initialization sequence of the subsystems it discovered.

13.2.4 Subsystem Initialization Sequencing

The subsystems' "init" routines are run sequentially based on the following two sequencing criteria, described in detail in the next sections:

- Subsystem Classes

The coarse sequencing criteria of entry point execution during system initialization based on the type of functionality to be performed by this subsystem. Types of subsystem classes (see Table 13-1 for the full list) include basic system services, interface driver code, upper layer protocols, and system management.

- Intra-Class Sequencing Property

The more granular method to order entry point execution within a given class.

13.2.5 Subsystem Classes

Subsystems are organized into classes. Classes provide a sorting order, which is primarily used for basic system software initialization dependencies. Classes might also be used to help delimit initialization completion points for system components that are required to suspend and synchronize with others before resuming initialization activities. In this case, early initialization for the component is accomplished in one subsystem class, and the remaining initialization activity completed in a subsequent subsystem class. For example, SUBSYS_CLASS_PRE_EHSA and SUBSYS_CLASS_EHSA allow initialization suspension and synchronization during startup of Active and Standby processors in High Availability implementations..

Each subsystem is assigned to a particular class when defined. The class designation is specified in the SUBSYS_HEADER() macro.

```
SUBSYS_HEADER(. . . , ulong class, . . .);
```

Table 13-1describes the current list of classes in the order in which their subsystems are started when the system is initialized.

Table 13-1 Subsystem Classes in Order of Initialization

Subsystem Class	Description
SUBSYS_CLASS_SYSINIT	A single subsystem run before all other subsystems. This subsystem calls the start_sysinit() function, which launches the init_process process, which performs the rest of the subsystem initialization.
SUBSYS_CLASS_REGISTRY	Registry subsystem class. Use this class for subsystems that contain only calls for the creation of a new registry. This allows registries to be started and defaults to be registered before any other subsystems start, which can prevent many simple initialization ordering problems.
SUBSYS_CLASS_KERNEL	Kernel subsystem class. Use this class for subsystems that are fundamental to the system software and must be initialized first. The most common usage is IOS system basics such as FLASH file system initialization, IPC Inter-Processor Control, Interface List management, and OIR Online Insertion & Removal. It may also be appropriate for media parser chains that will require extensions by the drivers in SUBSYS_CLASS_DRIVER.

Table 13-1 Subsystem Classes in Order of Initialization (continued)

Subsystem Class	Description
SUBSYS_CLASS_IFS	Additional file systems class. Use this class for initialization of any file systems that must run after all standard FLASH file systems (of SUBSYS_CLASS_KERNEL) have been initialized. For example, the C7500 RSP stores microcode for other components in additional file systems. Added in Release 12.2 for the C7500 RSP.
SUBSYS_CLASS_UCODE	Microcode download class. Use this class to download microcode to other system components. The microcode will reside in file systems already initialized within SUBSYS_CLASS_IKERNEL or SUBSYS_CLASS_IFS. Added in Release 12.2 for the C7500 RSP.
SUBSYS_CLASS_PRE_EHSA	Subsystems that must be initialized prior to the EHSA Enhanced High System Availability subsystem class, or that assure Active-Standby early initialization completion before beginning EHSA establishment. Examples are: Marvel AS5850 IPC InterProcessor Communications subsystem. Added in Release 12.3T for pre-EHSA initialization work.
SUBSYS_CLASS_EHSA	Enhanced High System Availability subsystem class. Use this class for subsystems that are required to establish the High Availability feature of IOS redundant platforms. Added in Release 12.1 for new EHSA implementation.
SUBSYS_CLASS_PRE_DRIVER	Pre-Driver class. Use this class for initialization that is performed in preparation for any interface driver setup. Examples are: restoring SNMP interface index numbers from NVRAM storage and SS7 Signaling System 7 protocol setup. New in Release 12.2 for SNMP support.
SUBSYS_CLASS_DRIVER	Driver subsystem class. Use this class for all varieties of media drivers, such as Ethernet, ATM, Flash, and so on.
SUBSYS_CLASS_PROTOCOL	Protocol subsystem class. Use this class for all network protocols, such as IP, DECnet, IPX, and so on.
SUBSYS_CLASS_LIBRARY	Library subsystem class. Use this class for subsystems that provide services to other subsystems. For example, the crypto libraries, VoIP services and some IOS File System manipulation tools are in this class.
SUBSYS_CLASS_MANAGEMENT	Management subsystem class. Use this class for management applications such as SNMP and parser subsystems.

When identifying the usage of the various classes, currently about 75% of all subsystems are defined in the driver, protocol, and management classes, with about 25% in each. The others are spread unevenly between the remaining classes with only a very limited number in the following special classes: IFS, ucode, pre_EHSA, EHSA, and Pre-Driver.

Note A good way to tell exactly what subsystem classes are supported on any particular level of IOS is the following user-level command: **show subsys class ?**

13.2.5.1 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.

Frequently, your subsystem will have dependencies on routines defined in other classes that will force your subsystem into a particular class.

13.2.6 Intra-Class Sequencing Property

The optional intra-class sequencing property is used to sequence subsystems within a particular class, providing refinement in system initialization entry point execution.

You may optionally define this property, specifying those subsystems of the same class as yours that must be initialized before your subsystem.

```
SUBSYS_HEADER( . . . , char *property);
```

If you have no sequencing requirements within your class, use a NULL as the last argument to the SUBSYS_HEADER() macro.

If there are subsystems within your class that, if they exist, must be initialized before your subsystem, define the “seq:” property. The subsystem names are separated by white space or commas. White space in and around subsystems names is ignored when parsing individual items. For more information on the sequencing property, see section 13.4.2, “Sequencing Property”.

13.2.6.1 Subsystem Sequencing Property Definitions: Example

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"
```

13.3 Compile-time Subsystems

Compile-time subsystems refer to the other usage of the term “subsystem”, which is more explicitly linktime subsystems. Linktime subsystems are defined within makefiles to define the independent code sections, which run during system initialization.

For example, see subsection 9.5.2.5, “Compile a Platform-Specific Subsystem” in the *Cisco Engineering Tools Guide*.

13.4 Subsystem Properties

Subsystem properties specify initialization dependencies. There are two types of properties:

- Sequencing—Defines the sequence in which subsystems must be initialized within a particular subsystem class. This is defined within the subsystem declaration in the SUBSYS_HEADER() macro in a .c file.
- Requirements—Defines the other subsystems required by this subsystem. These subsystems will be within other subsystem classes. This is defined within the makefile.

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.

13.4.1 Subsystem Property Definitions

By default, the sequencing within a particular subsystem class is caused by link order and is therefore unpredictable over time because link object specifications may change. If a particular subsystem requires other subsystems of the same class to be initialized before it, use the subsystem sequencing property.

Subsystem sequencing properties are defined in a header file. A subsystem property consists of a property identifier followed by one or more subsystem names. The property identifier is `seq:` for sequencing properties. The names are separated by white space or commas. White space in and around names is ignored when parsing individual items.

The sequencing needs are defined in the `SUBSYS_HEADER()` macro. This macro must appear in the source file that initializes the subsystem module. The `SUBSYS_HEADER()` macro sets up the subsystem header that is compiled into the data segment of the image.

The subsystem definition can contain one or two sequencing property lists. In 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:` property string, specify `NULL` for the property. Do not specify `seq:` alone with no subsystem name. This causes the code to do extra work to find out that there is nothing to process.

The requirements needs are defined in a makefile, such as the `makesubsys` file. The required subsystems must be in the system image for the specifying subsystem to operate. The required subsystems can be initialized before or after the specifying subsystem is initialized.

13.4.2 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 (see Table 13-1). The order of subsystem class startup is as follows:

- 1 `SUBSYS_CLASS_SYSINIT`
- 2 `SUBSYS_CLASS_REGISTRY`
- 3 `SUBSYS_CLASS_KERNEL`
- 4 `SUBSYS_CLASS_IFS`
- 5 `SUBSYS_CLASS_UCODE`
- 6 `SUBSYS_CLASS_PRE_EHSA`
- 7 `SUBSYS_CLASS_EHSA`
- 8 `SUBSYS_CLASS_PRE_DRIVER`

- 9 SUBSYS_CLASS_DRIVER**
- 10 SUBSYS_CLASS_PROTOCOL**
- 11 SUBSYS_CLASS_LIBRARY**
- 12 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.

13.4.2.1 Driver Initialization Sequence

The actual order of subsystems within the driver class is determined by slot/port sequence. Ordering can be enforced by the `seq:` field in the `SUBSYS_HEADER` macro for the driver class, but ordering between WAN and LAN should not be done and the `init` sequence should be avoided whenever possible. Ideally, you should write code that can be loaded and unloaded dynamically.

Note Incidental ordering is caused by link order, and you should count on that changing unannounced; for example, `$(DRIVERS)` disappears in the Reformation images.

13.4.3 Requirements Property

The requirements property defines the subsystems that must be present in order for this subsystem to operate. Required subsystems are expressed in link module requirements. All the subsystems listed in the makefile must be in the system image for the subsystem to operate. For example, use `sys/makesubsys` to define subsystems that remain consistent across all platforms, use `sys/makesubsys.platform` to define subsystems that remain consistent across all platforms, but cannot have its object files shared in a CPU family, and use `sys/obj-xx-xxxx/makesubsys.xxxx` to define subsystems that are platform-specific (the subsystem's object files are in the object directory for the platform). The subsystems listed in the requirements property can be in any subsystem class.

13.4.4 Error Messages

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

```

13.5 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.

```
SUBSYS_HEADER(char *name, ulong major_version, ulong minor_version,
              ulong edit_version, void *init, ulong class, char *property);
```

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 (subsys_type *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 requirements property in the makefile.

You define the subsystem requirements in the makefile. Add the following elements to the makefile to define the subsystem's requirements:

- `sub_name.desc`—string describing the subsystem
- `sub_name.link_req`—the subsystem's link requirements
- `sub_name.run_req`—the subsystem's runtime requirements
- `sub_name`—the subsystem's object files

Also, add the subsystem's dependencies. For example, to specify that `sub_cdp` is dependent on presence of the `sub_cdp.o` file, add:

```
sub_cdp.o : $(sub_cdp)
```

Note the different types of ‘subsystems’, the compile time subsystem (`sub_myname.o`) and the run time subsystem (`SUBSYS_HEADER` macro), where one type is described as a link subsystem meaning a link module defined in some `makefile.subsys` file where a bunch of object files are linked together in a single module, and then there is the IOS concept of a subsystem, with a `SUBSYS` header.

Conceptually (and practically), these different subsystem types are quite separate, and every link module does not need to have an IOS `SUBSYS` header. There can be quite complex interactions between the two types; for instance, lots of link modules that connect at link time with other modules - one of which may contain a `SUBSYS` header to allow initialization to take place, or multiple `SUBSYS` headers in a single link module to allow different elements to be initialized and connected to the system at distinctly different times. For example, in a single `sub_xxx` module, there may be portions that you want to initialize as a `KERNEL` subsystem, and other portions that you want to setup as a `DRIVER` module. Other kinds of `sub_xxx` modules may just be a set of platform specific functions that do not need a `SUBSYS` header.

13.5.1 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.

```
# define SNARK_MAJVERSION    1
# define SNARK_MINVERSION    0
# define SNARK_EDITVERSION   1

SUBSYS_HEADER (snark, SNARK_MAJVERSION, SNARK_MINVERSION, SNARK_EDITVERSION,
               snark_subsys_init, SUBSYS_CLASS_DRIVER, 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.

```
# define SNARK_MAJVERSION    1
# define SNARK_MINVERSION    0
# define SNARK_EDITVERSION   1

SUBSYS_HEADER (snark, SNARK_MAJVERSION, SNARK_MINVERSION, SNARK_EDITVERSION,
               snark_subsys_init, SUBSYS_CLASS_DRIVER, "seq: boojum,
               wibble");
```

13.6 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.

13.7 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

13.7.1 Create a New Subsystem

To determine whether to create a new subsystem, follow these steps:

- Step 1** 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 2** 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 `IPHOST`. Be sure all these requirements are in the makefile.
- Define the new subsystem by declaring the subsystem header, `SUBSYS_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 subsystem 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: $(2 * \text{case table size}) + 13$ 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
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

13.7.2 Rework System Processes

If not already previously completed, you must rewrite all older systems processes (at least five years old in 2002) to use the current Cisco IOS event-driven scheduler primitives. For example, 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 “Basic IOS Kernel Services” chapter.

13.7.3 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.

13.7.4 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.

Registries and Services

Added new section 14.5.1.2 “Incorrect Macro Usage”. (April 2011)

Added new section 14.19 “Manipulate SEQ_LIST Services”, which describes how to use the SEQ_LIST registry services. (November 2009)

Corrected the filename in section 14.5 “Steps to Create and Use a Registry” to read files_reg.mk in the subsection that describes how developers add or remove registries in the Registry ReDefine Project. (October 2009)

Updated the CASE type of service in section 14.1.2 “Types of Services” and section 14.10 “Manipulate CASE Services”. (April 2009)

Added SEQ_ILIST and SEQ_LIST to section 14.1.2 “Types of Services”. (March 2009)

14.1 Introduction

This chapter describes the programming interface and operation of Cisco IOS registries and services.

Note Cisco IOS registries and services development questions can be directed to the interest-os-registry@cisco.com alias.

The following topics are discussed in this chapter:

- Sample Registry Usage
- Registry Files
- The Registry Compiler
- Steps to Create and Use a Registry
- Registry Services
- Manipulate LIST Services
- Manipulate ILIST Services
- Manipulate PID_LIST Services
- Manipulate CASE Services
- Manipulate RETVAL Services
- Manipulate FASTCASE Services

- Manipulate LOOP Services
- Manipulate STUB Services
- Manipulate STUB_CHK Services
- Manipulate FASTSTUB Services
- Manipulate VALUE Services
- Manipulate CASE_LIST/CASE_LOOP Services
- Manipulate SEQ_LIST Services

14.1.1 Terms

registry

A collection of service points to install (register) and execute (invoke) callback functions and discrete values of PIDs. Each registry is frequently placed in a unique subsystem of type `SUBSYS_CLASS_REGISTRY`.

registry compiler

Preprocessor to generate function prototypes and definitions for all service points in all registries.

registry name

Defined in the `BEGIN REGISTRY REGISTRY_NAME` statement in the metalanguage. It is referenced when the registry is initialized in the `create_registry_registry_name()` function.

service

A type of service point, for example, CASE, LIST, or LOOP. However, in common usage, the term “service” really means “service point”.

service point

A single “managed function vector”, which allows for executing all, several, one, a default, or zero registered routines. Within Engineering, the term “service” is frequently used to mean “service point”.

service point name

Defined in the `DEFINE name` statement in the metalanguage. It is expanded to define the registry functions required for the `name` service point, such as `reg_add_name()` and `reg_invoke_name()`.

SUBSYS_CLASS_REGISTRY

Class of the subsystem that is initialized during network device startup and is used to create each registry using the `create_registry_registry_name()` function. For more on subsystem classes, see Chapter 13, “Subsystems.”

14.1.2 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.

Caselist

The run-time replacement for a list of callbacks that are executed sequentially.

Caseloop

The run-time replacement for a list of callbacks that are executed sequentially until one of the callbacks return a TRUE value.

LIST

A LIST service is the runtime replacement for a list of C functions that are executed sequentially. It reads through a list of functions, calling each function one function at a time in sequence.

ILIST

An intermittent list service is similar to a LIST service except that a special callback can be specified at invocation which is called between callbacks in the list. This ability is more commonly used to allow the invoking task to suspend on large lists that could otherwise become a CPU hog.

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 runtime replacement for a C switch statement with an implied break. This service reads through a list of functions until it finds the matching service.

RETNAL

A RETNAL service is identical to the CASE service except that it returns a value instead of a void.

FASTCASE

A FASTCASE service is basically a function vector table. There is no boundary checking performed on the indices passed.

FASTSTUB

A FASTSTUB service is a STUB service for fast switching.

LOOP

A LOOP service is a runtime replacement for a C while loop. Each function registered for the LOOP service is called until one of the functions returns TRUE.

SEQ_ILIST

A sequenced ILIST registry type. This allows you to specify a sequence for the ILIST registry.

SEQ_LIST

A sequenced LIST registry type. This allows you to specify a sequence for the LIST registry.

To control the order of the callback in the reg_invoke functions in a regular LIST registry, you must sequence the order of the reg_add functions. This is difficult to implement. As an example, suppose that in a LIST registry you have foo, bar, and foobar to be called, in that order. At runtime, you would have to call reg_add foo, then reg_add bar, and finally reg_add foobar. The problem is that the reg_add functions can come from different subsystems. If you have 100 callbacks, it is difficult to know the right sequence. With a SEQ_LIST, you do not have this constraint. In the above example, you can use SEQ_LIST to do reg_add foobar 3, reg_add foo 1, and reg_add bar 2. The numbers 3, 1, and 2 are the sequence numbers of the callbacks in the registry.

STUB

A STUB service takes zero or one functions (like a LIST service) and can return a value (like a RETNAL service).

STUB_CHK

A stub check (STUB_CHK) service is similar to a STUB service except that it protects against the unintentional overwriting of an existing stub.

VALUE

A VALUE service is a lookup table of 32-bit values.

14.1.3 Overview

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 various different types (see Section 14.1.2, “Types of Services”). 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.

Registries can be used to connect modules in a way that allows other kinds of APIs to correctly operate. One example is the plugin drivers, where a CASE registry is used to register a handler for a particular type of PA, and when the system finds this PA, that handler is invoked via the registry that creates a data object representing the PA; the data object itself then uses a different kind of API (a virtual function table) to interface with the system, but the original registry has been used to allow this data object to be built and attached to the system.

14.1.3.1 registry, service, service point

This information was originally in the “Cisco IOS Technical Note: Word of the Week” EDCS-184146 (2/4/02):

Historical Usage

In sys/Doc/Function_Registration, written in 1993, it says:

A registry is a collection of related services. We currently have the following well known registries:

Registry Name Nature of Services

=====

REG_SYS Operating System

REG_IP IP services
REG_MEDIA Encapsulation functions
REG_FAST Fast Switching

The well known registries are created in registry.c during system initialization....

A service is a collection of one or more C functions registered by various subsystems. Functions for a specific service will have the same calling and return conventions. The actual invocation of a service is referred to as a service point.

Current Usage

Andrew MacRae:

"Registry has two basic meanings. The first describes a specific IOS object that connects modules together: the LIST, CASE, RETVAL etc. things. Each instance of these I would call a 'registry', and at times I have also heard it called 'service'.

The second use of 'registry' describes a file which contains a *set* of the registries described earlier, but the use of this is usually related to a source code context, such as 'put that new entry in the media registry'. Generally, these separate files group together all the services for a particular module, and often the name of the file is used to describe the 'registry.' For example, 'the IP registry' would refer (depending on the context) to the file ip/ip_registry.reg or to the set of services in that file.

Sometimes I have heard people talk about the interface represented by a group of registries as a 'service'. I prefer to call that a 'module registry interface' or 'registry API', but even that is a little ambiguous.

The term 'service point' isn't in my general vocabulary when I talk about IOS, so I am loathe to really say what I think it means. However, perhaps it could refer to a single registry within a set of registries (not a bad definition), or perhaps it even refers to one endpoint of a module that is part of a registry service. In any case, because it is not in general use (at least not with me :-) I would prefer that the term not be used unless someone provides an accurate definition."

Michael Boe:

"My understanding is that 'registry service-point' is a single API within a registry (used in the file/set sense). So, consistent with Andrew's first definition above. At least, that's the way we've been using the term in the ION community.

Perhaps it even refers to one endpoint of a module that is part of a registry service. In any case, because it is not in general use (at least not with me :-) I would prefer that the term not be used unless someone provides an accurate definition.

This definition refers to the runtime binding. I find that people refer to the provision end of the runtime-bindings as 'the `reg_add()` function pointer' or sometimes a 'registry service-point provider' or even 'registry service endpoint.' Though I wince when I hear that last one.

Not in general use, but worth considering, is 'registry service-point callout.' This comes close to describing the item without being long or misleading. The code has 'callback' but maybe the code's worth changing to reflect reality at some point."

Carl Schaefer:

"With respect to common usage, Andrew's description seems right. 'registry' is used to refer to a set of related service points, and also to individual service points themselves. The term 'service point' is not often used."

14.1.4 Common Uses Of Registries

This section gives some background on ways in which you want to create or use registries.

14.1.4.1 What Are the Two Common Uses of Registries?

The two most common uses of registries are:

- Peer-to-peer: a subsystem might choose to cooperate with another subsystem if it is present (since it might be absent in certain bundles and/or on certain platforms) by requesting services of its peer.
- Bottom-to-top: a low-level subsystem (such as a datalink protocol driver or signalling layer) might choose to provide “indications” of various interesting events that other subsystems (which it has carefully and rigorously avoided any knowledge of) might require. An example would be ISDN needing to know if a DSX1 subsystem has had a red-alarm so that it might tear down all calls on that PRI.

In the latter case, one often sees developers trying to put the cart before the horse by adding their private hook into DSX1 (for example) so that on a particular code path, the upper-layer “client” subsystem might get a notification. This is wrong, since no other potential clients might make use of that same “trigger-point”.

Ideally, the trigger-points would be general and well chosen, and clients would have a rich set from which to select, rather than a helter-skelter set that gets episodically augmented (but without any sort of vision as to how it might evolve, at least not from the perspective of the group sustaining the low-level subsystem).

This would, of course, require all groups to be more diligent in respecting the ownership boundaries of other groups, in particular the sustaining groups—which should be happening in any case.

14.1.4.2 Two Other Ways to Think of Registries

Two other ways to think of registries and the relationship between subsystems is:

- Which came first? The predecessor is more likely to export registries that newer subsystems can register handlers for. This limits churn in the older subsystem, and avoids exponential growth as new ‘client’ subsystems are added.
- Which is more essential? That is, if a ‘stripped down’ or minimalist image was generated, which subsystem is more likely to be present? This is the one that should export its trigger-points (via indications) to other subsystems.

In both cases, the ‘exporting’ subsystem defines the registries and has the `reg_invoke_name()` sequences in its own code. The ‘importing’ subsystem uses ‘`reg_add_name()`’ to request information from that subsystem.

This model applies equally to platform-specific handlers, that platform independent code may use. The platforms will come and go, but the subsystem remains.

14.1.4.3 Words of Guidance When Adding New Registry Entries

Some final words of guidance when adding new registry entries: do not let immediate requirements impose tunnel vision on you. Think in terms of generalized solutions. For instance, when adding a trigger-point for which some nascent client requires a notification of transition into a new state, don’t limit the parameter list to a handle to the object. Rather, provide both the old and new states (before and after the transition). It is easier to have the client ignore parameters for which it has no need, than to cause churn (with the associated testing and dangers of regression defects and sync conflicts)

when having to come back and add in this parameter later when another client increases the scope of this particular notification. Further, it liberates all clients from having to keep track of that object's state from transition to transition, which would involve pointless duplication of the same data which is already maintained by the exporting subsystem.

Similarly, if providing notifications for state transitions, provide them for *all* arcs. Clients may attach to the particular registry hooks that they require (such as up-to-down transition, but not unconfigured-to-configured), and the overhead on an unused registry (that is, one for which no handlers are registered) is minimal.

Do not allow clients to modify their arguments as ‘side-effects’. This may be especially deleterious when multiple clients receive a notification, and one client modifies an object before the other clients have a chance to see its current state. This will cause inconsistencies. For this reason, passing objects as ‘const’ is highly recommended.

Furthermore, the registry name shouldn't include the name of any specific platform or hardware component. Certainly a registry name should accurately reflect the function of the registry, but if the registry name is excessively specific, it's likely that the registry itself will not have much utility in general.

Another indication of “tunnel-vision” registry design is a high reliance on STUB registries. A STUB registry is usually the easiest solution because it directly solves an inter-module linking problem. However, due to the single-function nature of a STUB, it can only ever solve one problem at a time. Over the long term, redesigning to use a LIST, CASE or LOOP is usually worthwhile because it can accommodate future needs as well as the current one. A STUB registry should be considered a last resort, and the author is obligated to make a good-faith effort to use a more flexible registry type instead.

At the very least, a STUB_CHK—sometimes you want to have a default handler for something (say a function that generates names of DS0 channels, given a pointer to an hwidb), but on certain platforms that deviate from the majority, you might want to overwrite that function with one of your own. In this case, the platform would ignore the fact that STUB_CHK is telling you that a handler is already installed if it initializes later, or the platform-independent code would not install the handler if one was already there, if it initialized later.

For example:

```
static void hwidb_to_namestring (hwidb *hwidb, char *buffer)
{
    /* for most platforms, this is the default ... */
    sprintf(buffer, "%d/%d:%d", ...);
}
static void c5800_does_things_differently_hwidb_to_namestring (hwidb *hwidb,
    char *buffer)
{
    /* clobbers installed handler deliberately */
    sprintf(buffer, "%d/%d/%d:%d", shelf_id(), ...);
}
```

14.1.5 Default Functions

A registry service point's default function is executed if a callback function does not exist, that is, no code has called `reg_add_name()`, or if the registered service point has been deleted by `reg_delete_name()`. Each registry type has a specific default function, but you can provide a different default function for a particular service by using `reg_add_default_name()`.

Note The LIST, ILIST, PID_LIST and LOOP services don't have default functions. If no callbacks are registered, then `reg_invoke_name()` returns without executing any functions.

For more information about the default functions for each particular service, see sections:

- LIST Service's Default Function
- ILIST Service's Default Function
- PID_LIST Service's Default Function
- CASE Service's Default Function
- RETVAL Service's Default Function
- FASTCASE Service's Default Function
- LOOP Service's Default Function
- STUB Service's Default Function
- STUB_CHK Service's Default Function
- FASTSTUB Service's Default Function
- VALUE Service's Default Function

14.1.6 REMOTE Registries in IOS

Registries that have their scope defined as REMOTE for ION, have their scope and other attributes ignored in IOS since those attributes only have meaning in ION (the definition does not affect the behavior of the registry in IOS). Code for IOS is generated the same as it would be without those attributes; it is only placed in a different file (for example, `reg_name.ios.regc` for IOS and `reg_name.ion.regc` for ION, instead of `reg_name.regh` where registries without REMOTE tags end up). See the *Cisco IONization 101 Guide* for information on REMOTE registries in ION.

14.2 Sample Registry Usage

Samples of registry usage are the resolutions for problems such as "How to call alternate routines depending on the image or an input argument" and "How to call many varying routines without changing the calling code". For example, suppose you want to complete the following tasks:

- 1 While shutting down an interface, call all routines that want to know of this state change. Pass in the `hwidb` of the interface being deactivated.
- 2 Pass on a packet for further network layer (Layer 3) protocol work. Call only the routines appropriate for that L3 protocol (IP, IPX,...). Arguments are the L3 protocol type and a pointer to the packet buffer.
- 3 On a PING command, interpret the L3 address. The calling code does not know the L3 protocol type, such as IP or IPX, but the called routines can determine if it is their format.
- 4 Get the default MAC address for this network device. This can use a default IOS routine or be platform-dependent.
- 5 Convert IOS "linktype" to/from the IEEE designation. For example, `LINK_IPV6 = x0079 ; TYPE_IPV6 (IEEE) = x86DD.`

Each of these problems has an elegant registry solution, as follows:

- 1 In the shutdown example, all registered routines are called. Services that support this are types LIST, ILIST, and PID_LIST.
- 2 In the packet-processing example, a single routine is called depending on the L3 protocol type. Routines register based on the L3 protocol type. (In Cisco IOS this is called the "linktype".) When the registry is invoked, only the correct routine is called. Services that support this functionality are CASE, RETVAL, and FASTCASE.
- 3 For the PING command, we cannot use a CASE implementation because the calling code cannot differentiate. Instead, we can use a LOOP service, whereby each routine is called in sequence and the calling routine either processes and returns TRUE or returns FALSE, indicating it cannot process. Control returns to the calling routine when a TRUE return is encountered.
- 4 For the default MAC address function, only one routine is called: either a default or platform-specific routine. The STUB and STUB_CHK services are used to get the default MAC address for a device.
- 5 The IOS linktype-IEEE conversions each require an indexing function and the return of a hex value. This is implemented using the VALUE service.

14.2.1 System Registries

Table 14-1 lists some registries available in the system registry library, `libregistry.a`, which is linked into Cisco IOS images in the `sys/makelibs` makefile. System registry source files (`.reg` files) are maintained in the `sys/sys_reg` directory in the IOS source tree, with filenames of the form `sys_xxx_registry.reg`, each containing a registry definition with the registry name, `SYS_xxx`, and descriptions of the service points that registry offers.

For example, the `sys_cdp_registry.reg` file describes the `SYS_CDP` registry, with service points that implement Cisco Discovery Protocol services such as `bridge_policy_cdp_denied` and `cdp_reset`.

See Section 14.3, “Registry Files”, and Section 14.4, “The Registry Compiler”, for details on registry source files and how service entry points are defined.

Table 14-1 Summary of Available System Registries

File Name	Registry Name	Description Summary of Service Points
<code>sys_acl_registry.reg</code>	<code>SYS_ACL</code>	Access List service points: Display, update, clear access list information; check access list numbers and whether a packet matches an access list.
<code>sys_arp_registry.reg</code>	<code>SYS_ARP</code>	ARP service points: Manage ARP table and cache entries; format ARP packets.
<code>sys_bflc_registry.reg</code>	<code>SYS_BFLC</code>	Failure logging service points: Send and display crash information.
<code>sys_boot_registry.reg</code>	<code>SYS_BOOT</code>	Boot service points: Display reason for booting and allow actions following bootloader completion.
<code>sys_bridge_registry.reg</code>	<code>SYS_BRIDGE</code>	Bridge service points: Manage bridging information and packets to be bridged.

Sample Registry Usage

Table 14-1 Summary of Available System Registries (continued)

File Name	Registry Name	Description Summary of Service Points
sys_bstun_registry.reg	SYS_BSTUN	Block Serial Tunnel (BSTUN) service points: Determine BSTUN prioritization and generate state change traps.
sys_cdp_registry.reg	SYS_CDP	Cisco Discovery Protocol (CDP) services: Manage CDP states upon IDB creation and encapsulation changes; discover CDP policy filtering.
sys_chksum_registry.reg	SYS_CHKSUM	Checksums: Calculate checksums on memory regions and comparison values.
sys_clns_registry.reg	SYS_CLNS	Connectionless Network Service (CLNS) service points: Manage CLNS routing information and packet status and updates.
sys_clock_registry.reg	SYS_CLOCK	Clock service points: Discover hardware clock and calendar status; clock backup; notify clients of clock status changes.
sys_comp_registry.reg	SYS_COMP	Component architecture support for platforms: Notify platform-specific code to prepare for and reload components.
sys_counters_registry.reg	SYS_COUNTERS	Counter service points: Manage interface counters.
sys_crash_registry.reg	SYS_CRASH	Crash-related support service points: Manage recording of crash information and notify clients of a core dump.
sys_decnet_registry.reg	SYS_DECNE	DECNET service points: Manage DECnet configuration, routing information, and conversion to/from CLNS.
sys_dhcp_registry.reg	SYS_DHCP	Dynamic Host Configuration Protocol (DHCP) service points: Activate DHCP client after system initialization and allow IP address assignment.
sys_ec_registry.reg	SYS_EC	EtherChannel (EC) service points: Set up ports for an EtherChannel interface.
sys_encaps_registry.reg	SYS_ENCAPS	Encapsulation service points: Show protocol-specific attributes from the encapsulation.
sys_fec_registry.reg	SYS_FEC	Fast EtherChannel (FEC) service points: Set up and manage Fast EtherChannel interface.
sys_fr_registry.reg	SYS_FR	Frame Relay service points: Indicate congestion notification for traffic shaping in Frame Relay packets.
sys_hostname_registry.reg	SYS_HOSTNAME	Hostname service points: Inform subsystems of hostname changes.

Table 14-1 Summary of Available System Registries (continued)

File Name	Registry Name	Description Summary of Service Points
sys_hw_registry.reg	SYS_HW	Hardware service points: Manage interface hardware conditions; notify subsystems of hardware status changes.
sys_ifs_registry.reg	SYS_IFS	IOS file system service points: Notify subsystems of file system status (device present, file system added or removed).
sys_ip_registry.reg	SYS_IP	IP service points: Manage IP address resolution and packet header translation.
sys_ipc_registry.reg	SYS_IPC	Interprocess Communication (IPC) service points: Notify IPC zone manager to reinitialize IPC slaves.
sys_lc_registry.reg	SYS_LC	Line Card (LC) service points: Manage LC core dumps and MAC encapsulation tables.
sys_lex_registry.reg	SYS_LEX	LAN extension (LEX) service points: Bind and unbind a LEX interface to and from a serial interface.
sys_log_registry.reg	SYS_LOG	Logging service points: Manage system logging and notify clients of various log-related events.
sys_name_registry.reg	SYS_NAME	Domain Name service points: Handle operations related to domain names (name lookup, matching, printing, and validating).
sys_netbios_registry.reg	SYS_NETBIOS	NETBIOS service points: Retrieve NETBIOS access lists and check access permissions.
sys_parser_registry.reg	SYS_PARSER	Parser service points: Handle command parsing and CLI prompt changes, and notify clients that a command was entered.
sys_platform_registry.reg	SYS_PLATFORM	Platform support service points: Manage platform initialization actions; determine platform features; print platform-specific information; prepare for core dumps.
sys_priority_registry.reg	SYS_PRIORITY	Priority service points: Manage and print priority list information.
sys_proc_registry.reg	SYS_PROC	Process service points: Manage and print process information; notify subsystems of process state changes.
sys_queue_registry.reg	SYS_QUEUE	Queue service points: Determine supported queuing methods and manage weighted fair queuing operations.
sys_rpm_registry.reg	SYS_RPM	Route Processor Module (RPM) service points: Handle RPM-specific notification and messages.

Table 14-1 Summary of Available System Registries (continued)

File Name	Registry Name	Description Summary of Service Points
sys_serial_registry.reg	SYS_SERIAL	Serial line service points: Manage certain serial interface operations.
sys_slave_registry.reg	SYS_SLAVE	Slave support service points: Handle slave processor operations, messages, and notifications.
sys_snapshot_registry.reg	SYS_SNAPSHOT	Snapshot routing service points: Handle snapshot state changes and manage related routing table updates.
sys_spd_registry.reg	SYS_SPD	Selective Packet Discard (SPD) service points: Update SPD state and send notification of SPD state changes to external processors.
sys_stun_registry.reg	SYS_STUN	Serial Tunnel service points: Generate or detect serial tunnel events.
sys_timer_registry.reg	SYS_TIMER	Timer service points: Set up and detect or notify clients of periodic events (one second, one minute, idle timeout, serial pulse).
sys_tty_registry.reg	SYS_TTY	TTY service points: Initialize and handle serial interface (TTY and VTY) operations, including formatting printf() strings.
sys_vciclass_registry.reg	SYS_VCCLASS	Vc-class service points: Initialize, trace, and monitor changes in VC Class parameters.
sys_vlan_registry.reg	SYS_VLAN	Virtual LAN (VLAN) service points: Manage VLAN interface settings in hardware and software IDBs.

There are several hundred system registry service points, some of which are documented in the *Cisco IOS API Reference*. Refer to the source file (.reg file) for a particular system registry to find the available service points and details, such as parameter lists, for services that you might need.

Most system registries are considered closed. If you can't find a service already defined with the functionality you need in one of the system registries, you can create your own registry or check with members of the interest-os-registry@cisco.com mailing alias for a consensus on whether to make an exception to add a new service point to an existing system registry. Use the guidelines in this chapter to determine if the services you need should be available to other subsystems, and are best implemented by building them into a registry.

14.2.2 Registry Usage Dependencies

Try to avoid having dependencies or ordering requirements among routines that have registered with a service. The order in which registered routines are invoked is usually determined by the order in which the service clients registered with the service point, especially for list-type registries (such as LIST, IILIST, and PID_LIST; see Sections 14.7 through 14.9), but order of registration depends on various system linking and subsystem initialization factors, and cannot be guaranteed.

If invocation of your registry service callback depends on the completion of another client's invocation, then rather than trying to order the registry callbacks, use other methods available in the Cisco IOS software to assure dependencies are preserved, such as:

- Implement a watched Boolean that interdependent routines can check to synchronize their operations (see Section 3.4.3.4, "Managed Booleans").
- Nest a call to a dependent callback routine within your callback handler, assuming that you know you have a dependency because you have knowledge of and access to the dependent function.

14.3 Registry Files

Table 14-2 describes the registry files associated with the Cisco IOS registries. You create some of these files, and others are created by the registry compiler.

Table 14-2 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 <code>thereg_name_registry.c</code> 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 <code>reg_invoke</code> , <code>reg_add</code> , 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 access 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 <code>SUBSYS_CLASS_REGISTRY</code> class, thus ensuring that the module is initialized prior to its use. The .c file #includes the registry .h and .regc files.

14.4 The Registry Compiler

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 are 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. The wrapper functions force full typechecking of registry call invocations. This typechecking is essential to prevent programming errors from producing subtle, elusive, and ultimately catastrophic bugs.

14.4.1 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.

14.4.2 .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 comments on a new line with the 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 interface_shutdown /* the function name is made up */
LIST
void
hwidb
END

DEFINE to_l3_protocol /* the function name is made up - it's really
                      raw_enqueue */

CASE
void
paktepe pak
MAX_L3_PROTO_TYPE /* really LINK_MAX_LINKTYPE
int L3_PROTOTYPE /* really LINKTYPE
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 LIST, ILIST, LOOP, PID_LIST, CASE, RETVAL, FASTCASE, STUB, STUB_CHK, or VALUE. For definitions of the service types, see Section 14.1.2, “Types of Services”.
- Next is the type declaration of the value returned by the service’s invocation function, `reg_invoke_`. For some registry services this is a fixed type. For LIST, ILIST, PID_LIST, and CASE services, the return type must be `void`. For LOOP services, it must be `boolean`. And for VALUE services, it must be `ulong`.
- The next item is a parameter list for the service invocation function prototype. If no parameters are required, enter a hyphen (-).
- After the invocation function parameters, service specific parameters are defined.

CASE, RETVAL, FASTCASE, and VALUE services require two additional items. The first is the number of cases for the case registry, and the second is a variable declaration used for indexing within the case. This variable declaration is prepended to the service invocation’s function parameter list.

PID_LIST services require one additional item. The item is the message identifier to be sent to each process when the PID_LIST service is invoked.

14.4.2.1 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
/*
 * An intermittent list service
 */
ILIST
void
-
END
```

```
DEFINE sample_service3
/*
 * 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_service4
/*
 *A STUB service
 */
STUB
    void
    int count, char *name
END

DEFINE sample_service5
/*
 * A robust STUB service
 */
STUB_CHK
    void
    int count, char *name
END

DEFINE sample_service6
/*
 * A CASE service
 */
CASE
    void
    boolean onoff, int no_packets
    MAX_CASES
    ushort protocol
END

DEFINE sample_service7
/*
 * A RETVAL service
 */
RETVAL
    boolean
    char * str, int errors, int drops, int collisions, int transmits
    MAX_CASES
    ulong media
END
```

```
DEFINE sample_service8
/*
 * A FASTCASE service
 */
FASTCASE
    boolean
    char * str, int errors, int drops, int collisions, int transmits
    MAX_FASTCASES
    ulong interface
END

DEFINE sample_service9
/*
 * A VALUE service
 */
VALUE
    ulong
    ulong value
    MAX_VALUES
    long type
END

DEFINE sample_service10
/*
 * A PID_LIST service
 */
PID_LIST
    void
    idbtype swidb
    MSG_SERVICE7
END

END REGISTRY
```

14.4.3 .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
```

14.4.4 .c File Contents

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 (subsstype *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);
```

14.5 Steps to Create and Use a Registry

Use the following steps to create and use a registry:

Step 1 Create a *reg_name.reg* file to define the registry and service points within. Details on the metalanguage and defining a registry and the included service points can be found in Section 14.4.2, “.reg File Metalanguage”. The general format is:

```
BEGIN my_reg_name

DEFINE my_case_svcpt
/* metalanguage defining my_case_svcpt */
END

DEFINE my_list_svcpt
/* metalanguage defining my_list_svcpt */
END

DEFINE my_next_svcpt
/* metalanguage defining my_next_svcpt */
END

/* additional service points */
END REGISTRY
```

Step 2 It is recommended that you explicitly create your registry. Strictly speaking, registry creation is only required if:

- (a) you want the registry to show up in the **show register** display command
and/or
- (b) you want defaults assigned.

To create your new registry, define a new registry subsystem. The subsystem initialization routine simply creates your new registry. Details on establishing your new registry can be found in Section 14.4.4, “.c File Contents”.

```
static void my_registry_init (subsysstype *subsys)
{
    create_registry_my_reg_name();
}

SUBSYS_HEADER(my_registry, . . . ,
              my_registry_init, SUBSYS_CLASS_REGISTRY,
              NULL);
```

Step 3 The Registry ReDefine Project changed the way developers add and remove registries in 12.2S and 12.3T PI06. Both ways are described here:

After 12.2S and 12.3T PI06 Registry ReDefine

The *make-lib/files_reg.mk* file defines the **FILES_REG** macro. The registry file path names from the directory level below *sys* are listed in alphabetical order in the definition of **FILES_REG** in *make-lib/files_reg.mk*.

Steps to Create and Use a Registry

To add registries, you must add registry file path names in alphabetical order directly into the definition of FILES_REG in make-lib/files_reg.mk. To remove registries, you must remove registry file path names directly from the definition of FILES_REG in make-lib/files_reg.mk.

For example, if you wanted to add const/native/env_registry.reg in IOS after the Registry Redefine Project, you would add the registry file above const/native/eobc_mcast_registry.reg, as was done here:

```
FILES_REG := \
...snip...
    const/catos-rp/scp_registry.reg \
    const/cwtlc/cwtlc_line_registry.reg \
    const/cygnus/cygnus_oir_registry.reg \
    const/native/capi_registry.reg \
    const/native/const_ipc.reg \
    const/native/env_registry.reg \
    const/native/eobc_mcast_registry.reg \
    const/native/oir_registry.reg \
    const/native/scp_dnld_registry.reg \
...snip...
```

Note You must add the specific path in the definition of the FILES_REG and every path must be listed in alphabetical order.

Before 12.2S and 12.3T PI06 Registry ReDefine

The make-lib/registries.mk contains a search path encoded in the FILES_REG variable. The approach is that developers should add wildcard patterns to the FILES_REG variable in make-lib/registries.mk.

The approach before the Registry ReDefine Project looks like the following example:

```
FILES_REG := $(wildcard $(SYSROOT)/*/*.reg \
$(SYSROOT)/os/*/*.reg \
$(SYSROOT)/c3800/*/*.reg \
$(SYSROOT)/toaster/*/*.reg \
$(SYSROOT)/voip/*/*.reg \
$(SYSROOT)/mm/*/*.reg \
$(SYSROOT)/cns/*/*/*.reg \
$(SYSROOT)/h323/*/*.reg \
$(SYSROOT)/switch/*/*.reg \
$(SYSROOT)/const/*/*.reg) \
```

The \$(wildcard) is a **make** function to scan the filesystem for patterns like \$(SYSROOT)/*/*.reg. This example pattern directs **make** to search all subdirectories of \$(SYSROOT) (which will be the sys directory) for all files that end with .reg.

The wildcard pattern for these files should be located in \$(SYSROOT)/make-lib/registries.mk for branches descending from florida, (for example, flo_isp, georgia,...). A similar method was used in older branches, (conn_isp, delaware) but the \$(wildcard) pattern was located elsewhere.

Note If you add a .reg file in a directory one level below sys, it is handled by the pattern */*.reg. However, if your directory structure is deeper than that, you need to add the specific path in the FILES_REG variable.

- Step 4** Add default functions as necessary. This is particularly useful for CASE functions. For example, on packet processing, if an L3 routine does not exist, drop the packet. For example, see Section 14.10.2.1, “Example: Add a Default Case Function”.

```
some_init_routine()
{
    reg_add_default_my_case_svc(arguments)
    . .
}

my_default_routine(arguments)
{
    /* code for default routine */
}
```

- Step 5** Code the routine additions (reg_add), and code the functions themselves. For example, see Section 14.10.1.2, “Example: Add a CASE Service”.

```
some_init_routine()
{
    reg_add_my_case_svc(. . ., my_case_svc_pt_for_ip)
    . .
}

my_case_svc_pt_for_ip(arguments)
{
    /* code for this routine */
}
```

- Step 6** Code the invocations of those routines:

```
some_runtime_routine()
{
    reg_invoke_my_case_svc(arguements)
    . .
}
```

14.5.1 Problems with Registry Files

The following sections describe how to avoid common problems with registries.

14.5.1.1 Missing Definition

After a .reg file has been changed, be sure to update all the registry files, for example:

```
make all_registry_files
```

Steps to Create and Use a Registry

A typical error for a build failing due to a missing definition of a `reg_xxx` function might be

```
.. /wan/comp_sys.c: In function `ppp_compression_config':
.. /wan/comp_sys.c:1933: warning: implicit declaration of function
      reg_invoke_if_reject_ppp_hw_comp'
```

Occasionally this error occurs because the definition of a function was actually removed from a registry. However, often it occurs because the registry has been changed but the .c and .h files have not been updated. To prevent or solve this problem, run **make dependencies** in the `sys` directory. If the problem persists, run **make superclean; make dependencies** or (more drastic, since it will delete all view private files) **cleartool lsprivate -other | xargs rm -f; make dependencies**.

14.5.1.2 Incorrect Macro Usage

You try to define a macro function that takes in a function and a character string as arguments. For example:

```
#define
rc_shim_reg_add_mplsvpnmb_get_next_vrf_name(mplsvpnmb_get_next_vrf_name, "mp
lsvpnmib_get_next_vrf_name") \
reg_add_mplsvpnmb_get_next_vrf_name(mplsvpnmb_get_next_vrf_name, "mplsvpnmb
_get_next_vrf_name")
```

You get a compile error when you try to compile the code:

```
In file included from .. /VIEW_ROOT/cisco.comp/routing/l3vpn-
svcs/mib/src/mplsvpnmb.c:23:
.. /VIEW_ROOT/ios/sys/h/.../routing/rc_mplsvpnmb_shim.h:18:83:
"mplsvpnmb_get_next_vrf_name" may not appear in macro parameter list
```

You may wonder why this macro is not able to accept a string as a parameter list. Macro parameters in the definition (not invocation) have to be C identifiers and not literals. The idea is that they are substituted by actual parameters during invocation. At the time of invocation, literals may be provided.

So, you would need to do the following:

```
#define rc_shim_reg_add_mplsvpnmb_get_next_vrf_name(func_name) \
reg_add_mplsvpnmb_get_next_vrf_name(func_name, #func_name)
```

14.5.2 Placement of `xxx_registry.o` in Makefiles

The `xxx_registry.o` modules must be packaged where they will be included in images with *all* clients - whether `reg_add`, `reg_invoke` or other. If there is one single subsystem that will be included in all images that make any `reg_add_xxx()`, `reg_invoke_xxx()`, `reg_<whatever>_xxx()` calls, and if you are pretty sure that this situation will not change in the future as more and more subsystems begin to reference this registry, then it is appropriate to include `xxx_registry.o` in that subsystem. Otherwise, it is usually more appropriate to edit `sys/makelibs` to include `xxx_registry.o` in `libregistry` rather than try to come up with a set of mutually-exclusive subsystems to which you can add `xxx_registry.o`.

Note For ION, follow the *Cisco IONization 101 Guidelines*, subsection 6.5.1, Step 4.

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. However, use of the registry library is discouraged when possible.

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 explicitly 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.

14.6 Registry Services

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 enqueueing 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.

14.6.1 Service Types Usage Guidelines

These guidelines are provided to help you understand some of the limitations of service type usage, particularly `STUB` services.

14.6.1.1 Limitations of STUB Service Type

Ideally software modules are independent building blocks that can be combined differently as needed. Of course these otherwise unconnected blocks need to be able to pass control and data between themselves. A registry is one way to do this. A `STUB` service is an extremely limiting mechanism because it allows only one handler function to be registered per built image. In many cases this may seem like all that's needed, but experience has shown that when a `STUB` appears to be the right answer, the problem is almost certainly not being looked at in a sufficiently general way.

The registry types that better allow for modular software development are the `LIST`, `CASE`, `LOOP`, etc. because they provide an extensible mechanism for connecting software blocks together. A `STUB` is not extensible, it is really a modularity violation with wrapper that allows it past the linker. To convert a `STUB` into something more useful, try to generalize the problem in some way. Consider other interface types, platforms, protocols, applications, etc. The odds are good that the problem you're solving is not truly unique, especially if you solve it right. At the end of it all (including thorough design review) you may not have found a way to generalize the `STUB`, but a good try is warranted. `STUB` should be the registry of last resort.

14.6.1.2 Comparison of Service Type Functionality

Cisco IOS has a “core” infrastructure for basic networking functions, and then has various points where feature-specific code can “hook in.” If you are familiar with Emacs and ELISP programming, you will recognize the idea of a hook. The core code provides the registries, and makes the `reg_invoke()` calls at various places in its execution. You add extra functionality to the core with `reg_add()` to registries so that your code can get called when the core makes the `reg_invoke()` calls.

So, the different kinds of registries look like:

- STUB—only one additional feature can register at this invocation point
- LIST—many additional features can register at this invocation point, and *all* will get to execute in sequence, in the order they were added to the registry
- LOOP—many additional features can register at this invocation point, and they execute until one says that the loop should terminate
- CASE—many can register but only one is invoked based on some attribute of the data
- CASE_LIST—this is just like a CASE registry, but for each case element, multiple services can be registered (equivalent to a LIST registry for each of the case element values). All functions registered for the case will get to execute.
- CASE_LOOP—this is just like a CASE registry, but for each case element, multiple services can be registered (it’s as if there is a LOOP registry for each of the case element values). All functions registered for the case will get to execute until one of them says that the loop should terminate.

In most cases it makes sense to plan for the system to be extensible in the future, so choosing to use the STUB service would be short-sighted in assuming that only one feature will ever want to be registered for this invocation point at a time. Just in case future features might be needed, you can use LIST instead of STUB, unless the nature of the function definitely indicates only one function to be registered at a time.

There are a few cases where using STUB makes sense. For example, if there is an image that runs on multiple platforms, and needs one and only one platform-specific routine to be registered for some invocation point, then a STUB is one possible solution. If the problem is one of abstracting platform characteristics, though, adding a CASE to either `platform_get_value()` or `platform_get_string()` could be a more appropriate solution.

14.6.2 show registry Support

The format of the `show registry` command output 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. The most common use of this output is to cut and paste the addresses 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 `reg_name_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 `reg_name_registry.reg` file.

14.7 Manipulate LIST Services

A LIST service is the runtime 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 each function one at a time in the order they were added to the list. Each `reg_add_name()` call to a LIST service appends the new function to the end of the list.

14.7.1 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);
void reg_used_name(void)
```

The registry compiler substitutes the service name declared on the `DEFINE` line of the service definition for `name` in the above functions and in the function prototype names. The registry compiler also uses the `arguments` declared in the service definition for the prototype of the invocation wrapper and callback.

14.7.1.1 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);
boolean reg_used_fast_setup(void);
```

14.7.1.2 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 in the list sequence as the service invocation traverses the list.

14.7.1.3 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);
```

14.7.1.4 Example: Inquire on Registrations of a LIST Service

Before adding a LIST service or invoking one, you can determine if a routine has been added. The return on this function is TRUE if at least one LIST service function has been registered or FALSE if no function has been registered. In our example, `reg_used_fast_setup()` is defined as:

```
static inline boolean reg_used_fast_setup (void)
{
    if (_registry_sys_proto.fast_setup.list)
        return (TRUE);
    else
        return (FALSE);
}
```

So, for example, if some code calls:

```
reg_add_fast_setup(atalk_fastsetup, "atalk_fastsetup");
```

then `reg_used_fast_setup()` returns TRUE. On the other hand, if no code has called that, or if all the `reg_add_name()` calls have been undone by `reg_delete_name()` calls, then `reg_used_fast_setup()` returns FALSE. This wrapper just tells whether there are *any* functions (one or more) registered for that service.

14.7.2 LIST Service's Default Function

If no service points have been added to a LIST registry, that is, no code has called `reg_add_name()`, or if all the registered service points have been deleted by `reg_delete_name()`, then the `reg_invoke_name()` call will return without executing any functions.

14.8 Manipulate ILIST Services

An intermittent list (ILIST) service is similar to a LIST service except it calls a provided function in-between calls to the LIST service.

14.8.1 Define an ILIST Service

To define an intermittent list service for a registry, use the following syntax:

```
DEFINE name
    ILIST
    void
    arguments
    END
```

The return value from an intermittent 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 intermittent list service definition, the registry compiler generates all the wrappers needed to manipulate the registry service. The following wrapper functions are generated for the service:

```
void reg_invoke_name(void (*_icallback)(void), 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.

14.8.1.1 Example: Define an ILIST Service

The following is an example of an intermittent list service definition. This service is called every one second by the `net_periodic_onesec` process. Functions registered with the service are generally called once per second. There are no arguments for the service's callbacks.

```
DEFINE onesec
ILIST
void
-
END
```

This intermittent list service definition generates the following wrappers:

```
void reg_invoke_onesec( void (*_icallback)(void) );
void reg_add_onesec(service_onesec_type callback, char *name);
void reg_delete_onesec(service_onesec_type callback);
```

14.8.1.2 Example: Add to an ILIST Service

When adding an intermittent list service for the `onesec` service, the registry compiler takes the parameters from the `reg_add_onesec()` function and uses them to generate a strongly typecast wrapper:

```
typedef void (*service_onesec_type) (void);

#define reg_add_onesec(a,b) _reg_add_onesec(a)
static inline void _reg_add_onesec (service_onesec_type callback)
{
    registry_add_ilist(callback, &registry_sys.onesec);
}
```

The following code uses the list service addition wrapper to add the `bgp_io_onesec()` function to the `onesec` service:

```
reg_add_onesec(bgp_io_onesec, "bgp_io_onesec");
```

Then, whenever the `onesec` service is invoked, the `bgp_io_sec()` function is called followed by a call to the function passed in the invocation function. (See next example.)

14.8.1.3 Example: Invoke an ILIST Service

When invoking an intermittent list service for the onesec service, the registry compiler takes the parameters from the `reg_invoke_onesec()` function and uses them to generate a strongly typecast wrapper:

```
void reg_invoke_onesec ( void (*_icallback)(void) )
{
    reg_ilist_struct *entry, *junk_node = NULL;
    leveltype         level;

    /* Entering Critical section. Need to protect ref_cnt increment */
    level = raise_interrupt_level (ALL_DISABLE);

    /* Prime the list service processing. */
    entry = _registry_sys.onesec;

    /* Check if List node is to be deleted, no need to
     * process it if so. The task currently referencing the node will
     * see about removing it from the list. */
    while(entry && (entry->delete_me != FALSE)) {
        entry = entry->next;
    }

    while (entry) {
        entry->ref_cnt++;      /* Lock list entry. */

        /* Leaving Critical Section. */
        reset_interrupt_level (level);

        /* See if memory needs to be freed. */
        if (junk_node != NULL) {
            registry_free_ilist_node( junk_node );
            junk_node = NULL;
        }

        /* Call list routine. */
        (*(service_onesec_type)entry->function) ();

        /* Call intermittent callback function between list callbacks. */
        if ( _icallback != NULL ) {
            (*_icallback)();
        }

        /* Entering Critical Section. Need to protect ulink &
         * reference cnt decrement. */
        level = raise_interrupt_level (ALL_DISABLE);

        /* Unlock entry & test reference count validity. */
        entry->ref_cnt--;      /* Unlock list entry. */
        if (entry->ref_cnt < 0) {
            crashdump (0);
        }

        /* Test to see if node should be deleted. */
        if ((entry->delete_me != FALSE) &&
            (entry->ref_cnt == 0)) {
```

```

/*
 * Unlink node from list. Remember list could
 * have change during last callback call.
 */
junk_node = entry;
if (junk_node->next != NULL) {
    junk_node->next->prev = junk_node->prev;
}
if (junk_node->prev != NULL) {
    junk_node->prev->next = junk_node->next;
}
else {
    /* Must be replacing head of list. (This could happen.) */
    _registry_sys.onesec = junk_node->next;
}
}

/* Grab next undeleted entry in list. If this node is marked to be
   deleted, its next & prev links are still valid. */
do {
    entry = entry->next;
} while (entry && (entry->delete_me != FALSE));
}

/* Leaving Critical Section.*/
reset_interrupt_level (level);

/* See if node needs to be freed.*/
if (junk_node != NULL) {
    registry_free_ilist_node (junk_node);
}
}

```

The following code illustrates how to invoke all the functions registered for the `onesec` service. It is also possible to invoke the registry by passing `NULL` for the intermittent function address:

```
reg_invoke_onesec (net_periodic_suspend);
```

14.8.2 ILIST Service's Default Function

If no service points have been added to an ILIST registry, that is, no code has called `reg_add_name()`, or if all the registered service points have been deleted by `reg_delete_name()`, then the `reg_invoke_name()` call will return without executing any functions.

14.9 Manipulate PID_LIST Services

A `PID_LIST` service is similar to a `LIST` service in that it is used as the runtime 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.

14.9.1 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. A maximum of 2 arguments are possible.

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.

14.9.1.1 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);
```

14.9.1.2 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_ADJUST.

14.9.1.3 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 code 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);
```

14.9.2 PID_LIST Service's Default Function

If no service points have been added to a PID_LIST registry, that is, no code has called reg_add_name(), or if all the registered service points have been deleted by reg_delete_name(), then the reg_invoke_name() call will return without executing any functions.

14.10 Manipulate CASE Services

A CASE service is a runtime replacement for a C switch statement with an implied break. This service reads through a list of functions until it finds the matching service.

14.10.1 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);
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.

Note Only one callback function can be registered at a time for each case in CASE registries. The *last* function registered will be the active one. To determine if a particular case is already registered, call the *reg_used_name()* function for that case.

14.10.1.1 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. The metalanguage will be defined from bottom to top.

For a CASE service, you must specify an index value of type int (in this example integer *linktype*, which is an indication of the network protocol on the link). The appropriate routine registered for that index value is the only routine run. If that index value does not have a routine registered and if a “default” routine is registered, that default routine is run.

Internally, a CASE service can be built using either of two formats:

- The first internal format is a multicolumn table of the format “index;routine”. This is the most memory-efficient method because entries are only created as routines are defined.

- Or, for better CPU efficiency, the array can be prebuilt to a maximum size with the routines being in the correct array position: routine-for-index-0; routine-for-index-1;... To implement the maximum-index format, you must supply the maximum-index immediately preceding the index definition. In our example the maximum-index is supplied as `LINK_MAXLINKTYPE`. This will take a little more memory but be much more CPU-efficient.

The invoked routines take only one argument, which is a pointer to the packet to be handled, `paktype *pak`.

The return code of any CASE service is `void` as specified in our example.

```
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);
```

14.10.1.2 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 code 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.

14.10.1.3 Example: Inquire on Registrations of a Particular CASE Value

Before adding a CASE service or invoking one, you can determine if a routine has been added for a particular index value. The return on this function is TRUE (this index value has a registered function) or FALSE (no function has been registered with this value). In our example, the `reg_used_raw_enqueue()` function is defined as:

```
static inline boolean reg_used_raw_enqueue(int linktype)
{
    return registry_case_used(linktype, &_registry_sys_proto.raw_enqueue);
}
```

The following code illustrates how to determine if a CASE service has been defined for a specified index. In this example, the code first determines if a function is registered for the `pak->linktype` value. If no function is registered, then the code calls `drop_packet()`. Otherwise, the code calls `reg_invoke_raw_enqueue()`, which invokes the registered function for this particular index.

```
if (!reg_used_raw_enqueue(pak->linktype)) {
    drop_packet(pak);
} else {
    reg_invoke_raw_enqueue(pak->linktype, pak);
}
```

For CASE registries, if a function has not been registered for the particular index and if `reg_invoke_name()` is called, where `name` is the actual name of the registry, then the default function will be executed. The `reg_used_name()` function allows you to avoid executing the default if no function is registered.

14.10.1.4 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);
```

14.10.2 CASE Service's Default Function

A registry service point's default function is executed if a callback function does not exist, that is, no code has called `reg_add_name()`, or if the registered service point has been deleted by `reg_delete_name()`. The default function is a function that does nothing and returns `void`:

```
void default_reg_return_void (void)
{
}
```

14.10.2.1 Example: Add a Default Case Function

Each registry type has a specific default function, but you can provide a different default function for a particular service by using `reg_add_default_name()`.

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.

14.11 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`. Although any value can be returned, typical values are Boolean TRUE/FALSE and integers (`int` or `uint`). The second example is an unusual one, where the return is a pointer within the packet.

Note Only one callback function can be registered at a time for each case in RETVAL registries. The *last* function registered will be the active one. To determine if a particular case is already registered, call the `reg_used_name()` function for that case.

14.11.1 Example 1: TRUE/FALSE Return Value with Indexed Array Structure

For this registry service, the index is `linktype` and it is formed internally as an array of size `LINK_MAXLINKTYPE`. A single argument is passed in (`idbtype_ *swidb`) and a Boolean of TRUE/FALSE is returned.

```
DEFINE proto_on_swidb
/*
 * Summary: Detect whether a protocol is running on an IDB.
 * Input: Switched on linktype.
 *         IDB of interest.
 * Returns: Return TRUE if a protocol is active on an IDB.
 */
RETVAL
boolean
struct idbtype_ *swidb
LINK_MAXLINKTYPE
int linktype
END
```

Functions generated are:

```

boolean reg_invoke_proto_on_swidb (int linktype,
                                   struct idbtype_ *swidb);

void reg_add_proto_on_swidb(int linktype,
                           service_proto_on_swidb_type callback,
                           char *textual_name);

void reg_add_default_proto_on_swidb(service_proto_on_swidb_type callback,
                                    char *textual_name);

boolean reg_used_proto_on_swidb(int linktype);

void reg_delete_proto_on_swidb(int linktype);

void reg_delete_default_proto_on_swidb(void);

```

14.11.2 Example 2: Return of Pointer, Smaller Non-indexed Array

This example also uses linktype as the index but internally this service uses the more memory-efficient implementation of [index1;function1];[index2;function2]. The single argument passed in is paktype_ *pak and the return is a pointer to the start of the compressed protocol header.

```

DEFINE compress_header
/*
 * Summary: Perform protocol-specific compression on the protocol header.
 * Input: Switched on linktype.
 *         Packet of interest.
 * Returns: Packet with protocol header compressed.
 */
RETVAL
struct paktype_ *
struct paktype_ *pak
0
int linktype
END

```

Functions generated are:

```

paktype_ * reg_invoke_compress_header (int linktype,
                                       struct idbtype_ *swidb);

void reg_add_compress_header(int linktype,
                            service_compress_header_type callback,
                            char *textual_name);

void reg_add_default_compress_header(service_compress_header_type callback,
                                     char *textual_name);

boolean reg_used_compress_header(int linktype);

void reg_delete_compress_header(int linktype);

void reg_delete_default_compress_header(void);

```

14.11.3 RETVAL Service's Default Function

A registry service point's default function is executed if a callback function does not exist, that is, no code has called `reg_add_name()`, or if the registered service point has been deleted by `reg_delete_name()`. The default function is one that does nothing and returns 0:

```
ulong default_reg_return_0 (void)
{
    return 0;
}
```

14.12 Manipulate FASTCASE Services

A FASTCASE service is a runtime replacement for a static function lookup table. This service indexes through a pre-allocated function table for improved performance. There is no range checking on the indices passed for lookup up functions. FASTCASE services are primarily used along the fastswitching execution path.

14.12.1 Define a FASTCASE Service

To define a FASTCASE service for a registry, use the following syntax:

```
DEFINE name
FASTCASE
return
arguments
maximum
index
END
```

The `return` line specifies the return type for the FASTCASE service.

The prototype of the variable used to index the CASE service is defined on the `index` line. The value define by `maximum` must be nonzero, so a lookup table can be generated to allow faster indexing for 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 FASTCASE service definition, the registry compiler generates all the wrappers needed to manipulate the FASTCASE service. The following wrapper functions are generated for a FASTCASE service:

```
void reg_invoke_name(index, arguments);
void reg_add_name(index, service_name_type *callback, char *textual_name);
void reg_delete_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`.

14.12.1.1 Example: Define a FASTCASE Service

The following is an example of a FASTCASE service definition for the `fr_pvc_switch_or_fs` service.

```
DEFINE fr_pvc_switch_or_fs
FASTCASE
boolean
paktype **pak
FR_TYPE_MAX
fr_type_path type
END
```

This CASE service definition generates the following wrappers:

```
#define reg_add_fr_pvc_switch_or_fs(a,b,c) _reg_add_fr_pvc_switch_or_fs(a,b)
void _reg_add_fr_pvc_switch_or_fs(fr_type_path type,
                                    service_fr_pvc_switch_or_fs_type callback)
void reg_delete_fr_pvc_switch_or_fs(fr_type_path type)
boolean reg_invoke_fr_pvc_switch_or_fs(fr_type_path type, paktype **pak)
```

14.12.1.2 Example: Add a FASTCASE Service

When adding a CASE service for the `fr_pvc_switch_or_fs` service, the registry compiler takes the parameters from the `reg_add_fr_pvc_switch_or_fs()` function and uses them to generate a strongly typecast wrapper:

```
typedef void (*sertypedef boolean (*service_fr_pvc_switch_or_fs_type)
(paktype **pak);

#define reg_add_fr_pvc_switch_or_fs(a,b,c) \
    _reg_add_fr_pvc_switch_or_fs(a,b)
static inline void _reg_add_fr_pvc_switch_or_fs (
    fr_type_path type, service_fr_pvc_switch_or_fs_type callback)
{
    _registry_fs.fr_pvc_switch_or_fs_array[type] = callback;
}
```

The following code uses the FASTCASE service addition wrapper to add the `fr_fast_switching_path()` function to the `fr_pvc_switch_or_fs` FASTCASE service:

```
reg_add_fr_pvc_switch_or_fs(FR_SWITCHING, fr_fast_switching_path,
"fr_switching_path");
```

Then, whenever the `fr_pvc_switch_or_fs` service is invoked with an index of `FR_SWITCHING`, the `fr_fast_switching_path()` function is called.

14.12.1.3 Example: Invoke a FASTCASE Service

When invoking a CASE service for the `fr_pvc_switch_or_fs` service, the registry compiler takes the parameters from the `reg_invoke_fr_pvc_switch_or_fs()` function and uses them to generate a strongly typecast wrapper:

```
static inline boolean reg_invoke_fr_pvc_switch_or_fs (fr_type_path type,
paktype **pak)
{
    return (*_registry_fs.fr_pvc_switch_or_fs_array[type]) (pak);
```

The following code illustrates how to invoke a FASTCASE service for a specified index. In this example, if a function is registered at the index returned by `fr_decode_path_inline(pak)`, the function is called when this statement is executed. If no function is defined for a given index, a default function is called. There is no range checking performed on the index. It is the responsibility of the user to ensure the index is valid.

```
reg_invoke_fr_pvc_switch_or_fs(fr_decode_path_inline(pak), &pak)
```

14.12.2 FASTCASE Service's Default Function

A registry service point's default function is executed if a callback function does not exist, that is, no code has called `reg_add_name()`, or if the registered service point has been deleted by `reg_delete_name()`.

If the registry return type is `void`, then the default function is a function that does nothing and returns `void`:

```
void default_reg_return_void (void)
{
}
```

Otherwise, the default function is one that does nothing and returns 0:

```
ulong default_reg_return_0 (void)
{
    return 0;
}
```

14.13 Manipulate LOOP Services

A LOOP service is a runtime replacement for a C `while` loop. Each function registered for the LOOP service is called until one of the functions returns TRUE.

14.13.1 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.

14.13.1.1 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);
```

14.13.1.2 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.

14.13.1.3 Example: Invoke a LOOP Service

When invoking 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:

```
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 `ip_udp_input` 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;
```

14.13.2 LOOP Service's Default Function

If no service points have been added to a LOOP registry, that is, no code has called `reg_add_name()`, or if all the registered service points have been deleted by `reg_delete_name()`, then the `reg_invoke_name()` call will return FALSE without executing any functions.

14.14 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. A STUB service is an extremely limiting service because it allows only one handler function to be registered per built image. For guidelines about using other service types, see Section 14.6.1, “Service Types Usage Guidelines”.

When it is tempting to use STUB services, there are alternatives. For example, instead of using STUB services to solve a platform override of a generic function, use the LOOP registry in the solution as follows:

```
if (!reg_invoke_platform_handles_xxx(x, y, z)) {
    generic_xxx(x, y, z);
}
```

This solution lends itself to be extended in the future, for more conditions of special handling and multiple handlers that may be present in the image in multiple combinations.

14.14.1 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.

Note When creating a new STUB registry (which is already described as something that should be avoided) use STUB_CHK instead. Only if STUB_CHK or one of the other registry services is clearly not appropriate should a STUB registry be used.

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(void);
boolean reg_used_name(void);
```

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.

14.14.1.1 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);
boolean reg_used_log_config(void);
```

14.14.1.2 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.

14.14.1.3 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);
```

14.14.1.4 Example: Inquire on Registrations of a STUB Service

Before adding a STUB service or invoking one, you can determine if a routine has been added by calling the `reg_used_name()` function. The return on this function is TRUE if a STUB service function has been registered or FALSE if no function has been registered.

The following code illustrates how to determine if a STUB service has been defined. In this example, the code first determines if a function is already registered for the STUB registry. It only adds a function if there is not one already defined.

```
if (!reg_used_log_config()) {
    reg_add_log_config(syslog_config, "syslog_config");
}
```

14.14.2 STUB Service's Default Function

A registry service point's default function is executed if a callback function does not exist, that is, no code has called `reg_add_name()`, or if the registered service point has been deleted by `reg_delete_name()`.

If the registry return type is `void`, then the default function is a function that does nothing and returns `void`:

```
void default_reg_return_void (void)
{
}
```

Otherwise, the default function is one that does nothing and returns 0:

```
ulong default_reg_return_0 (void)
{
    return 0;
}
```

14.15 Manipulate STUB_CHK Services

A STUB_CHK service is identical to a STUB service, except for more error checking in the `reg_add_name()` function.

14.15.1 Define a STUB_CHK Service

To define a stub check service for a registry, use the following syntax:

```
DEFINE name
STUB_CHK
return
arguments
END
```

The `return` line specifies the return value from a stub check 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 check service definition, the registry compiler generates all the wrappers needed to manipulate the stub check service. The following wrapper functions are generated for a stub check service:

```
void reg_invoke_name(arguments);
registry_status_e reg_add_name(service_name_type *callback, char
*textual_name);
void reg_delete_name(void);
```

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.

14.15.1.1 Example: Define a STUB_CHK Service

The following is an example of a stub check service definition. This service is called to initialize the log facility in the router.

```
DEFINE ipfib_table_update_linecard
STUB_CHK
void
parseinfo *csb
END
```

This stub check service definition generates the following wrappers:

```
void reg_invoke_ipfib_table_update_linecard(parseinfo *csb);
registry_status_e reg_add_ipfib_table_update_linecard(service_log_config_type
*callback, char *name);
void reg_delete_log_ipfib_table_update_linecard(void);
```

14.15.1.2 Example: Add to a STUB_CHK Service

When adding a stub check service for the `log_config` service, the registry compiler takes the parameters from the `reg_add_ipfib_table_update_linecard()` function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_ipfib_table_update_linecard(a,b) \
    _reg_add_ipfib_table_update_linecard(a)

static inline registry_status_e _reg_add_ipfib_table_update_linecard(
    service_ipfib_table_update_linecard_type
callback)
{
    if (_registry_ipfib.ipfib_table_update_linecard !=
&default_reg_return_void) {
        registry_add_stub_exception("reg_add_ipfib_table_update_linecard");

        return (REG_FAIL);
    }
    else {
        /* Okay to replace callback. */
        _registry_ipfib.ipfib_table_update_linecard = callback;

        return (REG_OK);
    }
}
```

The following code uses the STUB service addition wrapper to add the `syslog_config()` function to the `log_config` service:

```
reg_add_ipfib_table_update_linecard(ipfib_table_update_linecard,
"ipfib_table_update_linecard");
```

Then, whenever the `ipfib_table_update_linecard` service is invoked, the `ipfib_table_update_linecard()` function is called. If another function had been added previously an error message would be displayed on the console, and the add would have failed. To avoid this, a `reg_delete_ipfib_table_update_linecard()` can be called to reset the stub function back to the default.

14.15.1.3 Example: Invoke a STUB_CHK Service

When invoking a stub check service for the `log_config` service, the registry compiler takes the parameters from the `reg_invoke_ipfib_table_update_linecard()` function and uses them to generate a strongly typecast wrapper:

```
static inline void reg_invoke_ipfib_table_update_linecard (parseinfo *csb)
{
    (*_registry_ipfib.ipfib_table_update_linecard) (csb);
```

The following code illustrates how to invoke the function registered for the `ipfib_table_update_linecard` service:

```
reg_invoke_ipfib_table_update_linecard(csb);
```

14.15.2 STUB_CHK Service's Default Function

A registry service point's default function is executed if a callback function does not exist, that is, no code has called `reg_add_name()`, or if the registered service point has been deleted by `reg_delete_name()`.

If the registry return type is `void`, then the default function is a function that does nothing and returns `void`:

```
void default_reg_return_void (void)
{
}
```

Otherwise, the default function is one that does nothing and returns 0:

```
ulong default_reg_return_0 (void)
{
    return 0;
}
```

14.16 Manipulate FASTSTUB Services

A FASTSTUB service is a STUB service for fast switching that takes zero or one functions (like a LIST service) and can return a value (like a RETVAL service).

A FASTSTUB2 service is a STUB service for fast switching that takes two functions (like a LIST service) and can return a value (like a RETVAL service).

A FASTSTUB3 service is a STUB service for fast switching that takes three functions (like a LIST service) and can return a value (like a RETVAL service).

A FASTSTUB4 service is a STUB service for fast switching that takes four functions (like a LIST service) and can return a value (like a RETVAL service).

14.16.1 Define a FASTSTUB Service

To define a FASTSTUB service for a registry, use the following syntax:

```
DEFINE name
FASTSTUB
return
arguments
END
```

The *return* line specifies the return type for the FASTSTUB service.

On the *arguments* line, you specify the arguments for the service, and, as a consequence, the prototype of the functions called.

From the FASTSTUB service definition, the registry compiler generates all the wrappers needed to manipulate the FASTSTUB service. The following wrapper functions are generated for a FASTSTUB service:

```
void reg_invoke_name(arguments);
void reg_add_name(service_name_type *callback, char *textual_name);
void reg_delete_name(void);
boolean reg_used_name(void);
```

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.

14.16.1.1 Example: Define a FASTSTUB Service

The following is an example of a FASTSTUB service definition for the tbridge_forward service:

```
DEFINE tbridge_forward
/*
 * Transparently bridge a packet for either the RSP or LES environments;
 * 1) From an RSP interface to another RSP interface.
 * 2) From an LES interface to another LES interface.
 * Obsoletes les_bridge_receive.
 *
 * Returns TRUE if the packet was forwarded, flooded, or filtered,
 * in which case the received packet will have been freed.
 * Returns FALSE if the packet should be received by the bridge itself,
 * in which case the received packet will not have been freed.
 */
FASTSTUB
boolean
struct paktype_ *pak
END
```

This FASTSTUB service definition generates the following wrappers:

```
#define reg_add_tbridge_forward(a,b) \
    _reg_add_tbridge_forward(a)

static inline void _reg_add_tbridge_forward (
    service_tbridge_forward_type callback)
{
    _registry_fs->tbridge_forward = callback;
}

static inline boolean reg_invoke_tbridge_forward (
    struct paktype_ *pak)
{
    boolean rv;

    service_tbridge_forward_type function_ptr;

    function_ptr = (service_tbridge_forward_type)
    _registry_fs->tbridge_forward;
    REG_ELOG(EL_REG, EV_REGSVC, function_ptr);
    rv = (boolean) (*function_ptr) (pak);
    REG_ELOG(EL_REG, EV_REGSVCDN, function_ptr);
    return (rv);
}

static inline void reg_delete_tbridge_forward (void)
{
    _registry_fs->tbridge_forward = (void *)&default_reg_return_0;
}

static inline boolean reg_used_tbridge_forward (void)
{
    return (((void *)(_registry_fs->tbridge_forward) != 
        (void *)default_reg_return_0) &&
        ((void *)(_registry_fs->tbridge_forward) != 
        (void *)default_reg_return_void));
}
```

The following is an example of a FASTSTUB2 service definition for the l2x_ip_udp_input service:

```
DEFINE l2x_ip_udp_input
/*
 *  The point where UDP packets enter the L2X (level-2 forwarding)
 *  subsystem
 */
FASTSTUB2
boolean
struct hwidbtype_ *hwidb, struct paktype_ *pak
END
```

The only difference in the wrappers generated for FASTSTUB2 (& onwards) from FASTSTUB is in the definition of `reg_invoke_name()`.

For example:

```
static inline ulong reg_invoke_tbridge_cmf (
    struct paktype_ *pak,
    struct tbifd_type_ **flood_list)
{
    ulong rv;

    service_tbridge_cmf_type function_ptr;

    function_ptr = (service_tbridge_cmf_type) _registry_fs->tbridge_cmf;
    REG_ELOG(EL_REG, EV_REGSVC, function_ptr);
    rv = (ulong) (*function_ptr)(pak, flood_list);
    REG_ELOG(EL_REG, EV_REGSVCNDN, function_ptr);
    return (rv);
}
```

The following is an example of a FASTSTUB3 service definition for the `dma_output_pak_coalesce` service:

```
DEFINE dma_output_pak_coalesce
/*
 * This registry coalesces a particle-based packet into a
 * contiguous one using hardware assist (if applicable). When the
 * coalesce is complete, the completion function is called so that
 * further processing can be done on the packet. This function is
 * used in fastsend functions, so the IDB is the outbound
 * interface.
*/
FASTSTUB3
boolean
struct hwidbtype_ *idb, struct paktype_ *pak, void *func
END
```

14.16.1.2 Example: Add a FASTSTUB Service

When adding a FASTSTUB service for the `tbridge_forward` service, the registry compiler takes the parameters from the `reg_add_tbridge_forward()` function and uses them to generate a strongly typecast wrapper:

```
#define reg_add_tbridge_forward(a,b) _reg_add_tbridge_forward(a)

static inline void _reg_add_tbridge_forward (
    service_tbridge_forward_type callback)
{
    _registry_fs->tbridge_forward = callback;
}
```

The following code uses the FASTSTUB service addition wrapper to add the `tbridge_rsp_forward()` function to the `tbridge_forward` service:

```
reg_add_tbridge_forward(tbridge_rsp_forward, "tbridge_rsp_forward");
```

Note that:

```

*
* The following fields of the rsp_pak should be set up by the caller
* prior to the invocation of reg_invoke_tbridge_forward():
*     if_input
*     datagramsize
*     datagramstart
*     mac_start
*     addr_start
*     info_start
*     rxtype
*
```

Then, whenever the tbridge_forward service is invoked, the tbridge_rsp_forward() function is called.

14.16.1.3 Example: Invoke a FASTSTUB Service

When invoking a FASTSTUB service for the tbridge_forward service, the registry compiler takes the parameters from the reg_invoke_tbridge_forward() function and uses them to generate a strongly typecast wrapper:

```

static inline boolean reg_invoke_tbridge_forward (
    struct paktype_ *pak)
{
    boolean rv;

    service_tbridge_forward_type function_ptr;

    function_ptr = (service_tbridge_forward_type)
        _registry_fs->tbridge_forward;
    REG_ELOG(EL_REG, EV_REGSCV, function_ptr);
    rv = (boolean) (*function_ptr) (pak);
    REG_ELOG(EL_REG, EV_REGSCVDN, function_ptr);
    return (rv);
}
```

The following code illustrates how to invoke the function registered for the tbridge_forward service:

```

reg_invoke_tbridge_forward(pak);
return (TRUE);
```

14.16.2 FASTSTUB Service's Default Function

A registry service point's default function is executed if a callback function does not exist, that is, no code has called reg_add_name(), or if the registered service point has been deleted by reg_delete_name().

If the registry return type is void, then the default function is a function that does nothing and returns void:

```

void default_reg_return_void (void)
{}
```

Otherwise, the default function is one that does nothing and returns 0:

```
ulong default_reg_return_0 (void)
{
    return 0;
}
```

14.17 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.

14.17.1 Define a VALUE Service

To define a VALUE service for a registry, use the following syntax:

```
DEFINE name
VALUE

type
maximum
index
END
```

The return value from a VALUE service is always *ulong*.

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*.

14.17.1.1 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



```

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)
```

14.17.1.2 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.

14.17.1.3 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);
```

14.17.2 VALUE Service's Default Function

If a service point has not been added for a particular value in a VALUE registry, then the `reg_invoke_name()` call will return the registry service's default value. This value is 0. The `reg_add_default_name()` function can be used to specify a different default value.

Note If the VALUE registry is defined with a nonzero “maximum,” then the default behavior works somewhat differently. If `reg_invoke_name()` is passed a value < maximum and a specific return value has not been added for that input value, then a value of 0 will be returned. If `reg_invoke_name()` is passed a value \geq maximum, and a specific return value has not been added for that input value, then the registry’s default value will be returned.

14.17.2.1 Example: Add a Default Value

A VALUE registry, by default, will return 0 if a service point has not been added for the value that is passed to `reg_invoke_name()`. You can change the default value that is returned by using `reg_add_default_name()`.

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.

14.18 Manipulate CASE_LIST/CASE_LOOP Services

This feature has been committed to the 12.0S (conn_ips), kangaroo and, in June 2003, the 12.3T branches.

The new registry types, CASE_LIST and CASE_LOOP services, are aimed at solving the linear dependence of registry definitions on features for certain kinds of IOS applications, which otherwise could lead to some scalability and maintainability problems.

To better understand the need for the new registry types, let us consider two examples. For Example(1), consider the current CASE registry implementation in IOS. This can be easily represented as a C switch statement as follows:

```
Switch (tag) {
    Case val_1:
        Callback_a();
        Break;
    Case val_2:
        Callback_b();
        Break;
    ...
    Case val_m:
        Callback_n();
        Break;
    Default:
        Return_void();
        Break;
}
```

The above setup works fine as long as there is just one callback that needs to be invoked depending on the case value (tag). Now, let us consider Example(2), where there are multiple callbacks for each CASE value.

```
Switch (tag) {  
    Case val_1:  
        Callback_a1();  
        Callback_a2();  
        ...  
        Callback_n1();  
        Break;  
    Case val_2:  
        Callback_b1();  
        Callback_b2();  
        ...  
        Callback_n2();  
        Break;  
        ...  
    Case val_m:  
        Callback_m1();  
        Callback_m2();  
        ...  
        Callback_nm();  
        Break;  
    Default:  
        Return_void();  
        Callback_df1();  
        Callback_df2();  
        ...  
        Callback_ndf();  
        Break;  
}
```

With the existing registry infrastructure, the most logical way to implement Example(2), would be to define a List registry service for the set of callbacks in each case. This approach requires $m+2$ registry services; m list registry services for the m cases, 1 for the default case and 1 for the Case registry itself. So, as the value of m increases, the number of new registry services increases as $O(m)$. This implementation therefore does not seem to be very attractive for CASE registries, which have a large number of cases. The same problem exists if we required a LOOP registry functionality for the callbacks in each case value. Except now, we have to revert to using the LOOP and RETVAL registry services for our implementation, since the CASE registry does not return any values.

For problems like the one described in Example(2), it would be nice to have a solution, which requires $O(1)$ new registry services i.e., a solution independent of the number of cases. The proposed CASE_LIST and CASE_LOOP registries are a step in this direction and satisfy this requirement by having the LIST and LOOP registries as part of the underlying CASE and RETVAL registry infrastructure respectively. Note that both these new registry types do not affect the run time, when compared to the first solution which requires $O(m)$ new registry services.

In applications related to QoS, an additional requirement to the basic CASE_LIST/CASE_LOOP functionality mentioned in Example(2) arises in the form of a fallback option for each case value.

```
Switch (tag) {
    Case val_1:
        Callback/s ? Main list for val_1
        Break;
    Fallback val_1:
        Callback/s ? Fallback list for val_1
        Break;
    Case val_2:
        Callback/s
        Break;
    Fallback val_2:
        Callback/s
        Break;
        ...
    Default:
        Callback/s ? Default list
        Break;
}
```

For the CASE_LIST registry, the main list for the case value is checked. If not present, then the fallback list for the case value is checked before the default list. For the CASE_LOOP registry, if the main list is not present or returns false, the fallback list is checked before the default list.

For more complete information on the CASE_LIST services, please see EDCS-129132 “*An Introduction to the CASE_LIST and CASE_LOOP Services.*”

14.18.1 API Changes

14.18.1.1 Add a Callback Function to the Default List of Callbacks

To add “callback_function” to the default list of callbacks for the registry service “name”, call the [reg_add_default_name\(\)](#) function.

```
#include <foo registry.h>
boolean reg_add_default_name(callback_function,
                             "textual_reference_for_callback_function");
```

14.18.1.2 Add a Callback Function to the Fallback List of Callbacks

To add “callback_function” to the fallback list of callbacks for the tag specified by “index” in the registry service “name”, call the [reg_add_fallback_name\(\)](#) function.

```
#include <foo registry.h>
boolean reg_add_fallback_name(index,
                             callback_function,
                             "textual_reference_for_callback_function");
```

14.18.1.3 Add a Callback Function to the Main List

To add “callback_function” to the main (primary) list of callbacks for the tag specified by “index” in the registry service “name”, call the `reg_add_name()` function.

```
#include <foo registry.h>
boolean reg_add_name(index,
                      callback_function,
                      "textual_reference_for_callback_function");
```

14.18.1.4 Delete Callbacks from the Main List

To delete all the callbacks from the main (primary) list for the tag specified by “index” in the registry service “name”, call the `reg_delete_all_name()` function.

```
#include <foo registry.h>
boolean reg_delete_all_name(index);
```

14.18.1.5 Delete All Callbacks in the Default List

To delete all the callbacks in the default list for the registry service “name”, call the `reg_delete_default_all_name()` function.

```
#include <foo registry.h>
void reg_delete_default_all_name(void);
```

14.18.1.6 Delete Callback Function from Default List

To delete “callback_function” from the list of default callbacks for the registry service “name”, call the `reg_delete_default_name()` function.

```
#include <foo registry.h>
boolean reg_delete_default_name(callback_function);
```

14.18.1.7 Delete All Callbacks from the Fallback List

To delete all the callbacks from the fallback list for the tag specified by “index” in the registry servide “name”, call the `reg_delete_fallback_all_name()` function.

```
#include <foo registry.h>
boolean reg_delete_fallback_all_name(index);
```

14.18.1.8 Delete the Callback Function from the Fallback List

To delete the “callback_function” from the fallback list of callbacks for the tag specified by “index” in the registry service “name”, call the `reg_delete_fallback_name()` function.

```
#include <foo registry.h>
boolean reg_delete_fallback_name(index,
                                 callback_function);
```

14.18.1.9 Delete the Callback Function from the Main List

To delete the “callback_function” from the main (primary) list of callbacks for the tag specified by “index” in the registry service “name”, call the [reg_delete_name\(\)](#) function.

```
#include <foo registry.h>
boolean reg_delete_name(index,
                        callback_function);
```

14.18.1.10 Invoke the Registry Service Name for Specified Tag

To invoke the registry service “name” for the specified tag “index”, call the [reg_invoke_name\(\)](#) function.

For CASE_LIST:

```
#include <foo registry.h>
void reg_invoke_name(index,
                      arguments);
```

For CASE_LOOP:

```
#include <foo registry.h>
boolean reg_invoke_name(index,
                        arguments);
```

14.18.1.11 Check if Specified Tag is Currently Being Used

To check if the specified tag “index” is currently being used by the registry service “name” i.e., either a primary or fallback list for the tag exists, call the [reg_used_name\(\)](#) function.

```
#include <foo registry.h>
boolean reg_used_name(index);
```

14.19 Manipulate SEQ_LIST Services

SEQ_LIST is a sequenced LIST registry type. It allows you to specify a sequence for the LIST registry.

To control the order of the callback in the `reg_invoke` functions in a regular LIST registry, you must sequence the order of the `reg_add` functions. This is difficult to implement. As an example, suppose that in a LIST registry you have `foo`, `bar`, and `foobar` to be called, in that order. At runtime, you would have to call `reg_add foo`, then `reg_add bar`, and finally `reg_add foobar`. The problem is that the `reg_add` functions can come from different subsystems. If you have 100 callbacks, it is difficult to know the right sequence. With a SEQ_LIST, you do not have this constraint. In the above example, you can use SEQ_LIST to do `reg_add foobar 3`, `reg_add foo 1`, and `reg_add bar 2`. The numbers 3, 1, and 2 are the sequence numbers of the callbacks in the registry.

SEQ_LIST is similar to the LIST registry except that you add a sequence number after the `reg_add` function name.

Here is the current SEQ_LIST registry implementation in IOS:

```
enum {
    FIRST_CLIENT = 1,
    SECOND_CLIENT,
    ...
    LAST_CLIENT
} client_seq_t;

reg_add_test_function ( SECOND_CLIENT, second_client_callback_fn,
"function_name");
```

Then when calling the registry, it is the same as with the LIST registry, for example, the `reg_invoke_test_function(...);`

The above code shows that SEQ_LIST only needs to specify a sequence number when doing `reg_add`. Then when `reg_invoke` is called, the callback function will be in sequence according to the `client_seq_t` order.

Time-of-Day Services

15.1 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.

Note Cisco IOS time-of-day services questions can be directed to the interest-os@cisco.com alias.

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.

15.1.1 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_ {
    ulong epoch_secs;           /* Seconds */
    ulong epoch_frac;          /* Fractional seconds */
} clock_epoch;
```

15.1.2 Time Formats

The Cisco IOS software uses three different time formats, which are referred to by the following names:

- `clock_epoch` structure
- UNIX format
- `ios_timeval` structure

Note Starting in releases 12.2S and 12.4T, the name of the Cisco IOS time-of-day `timeval` structure was renamed `ios_timeval` (see `COMP_INC(kernel, clock.h)`) to prevent conflict with the POSIX `timeval` structure definition (see `COMP_INC(posix, sys/time.h)`) after some POSIX library functions were added to the common code base.

15.1.2.1 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 year 2036.

15.1.2.2 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.

15.1.2.3 ios_timeval Structure

The `ios_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 ios_timeval_ {
    ulong year;           /* Year, AD (1993, not 93!) */
    ulong month;          /* Month in year (Jan = 1) */
    ulong day;            /* Day in month (1-31) */
    ulong hour;           /* Hour in day (0-23) */
    ulong minute;         /* Minute in hour (0-59) */
    ulong second;         /* Second in minute (0-59) */
    ulong millisecond;   /* Millisecond in second (0-999) */
    ulong day_of_week;   /* Sunday = 0, Saturday = 6 */
    ulong day_of_year;   /* Day in year (1-366) */
    long tz_offset;       /* Time zone offset (seconds from UTC) */
} ios_timeval;
```

15.1.3 System Clock: Description

The heart of the Cisco IOS time-of-day services is the *system clock*. The system clock, or “wall clock time,” 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.

The system clock or the wall clock time is affected by NTP and CLI. For more information about NTP, see section 15.1.5, “Network Time Protocol.”

Note Managed timers are based on passive timers, which are not affected by NTP and CLI. For more information about managed timers, see section 16.4, “Managed Timers” in the “Timer Services” chapter.

Note If you want your application to adjust its passive timers based on changes in wall clock time, you can invoke the `clock_set_notify()` registry service from the clock code. For more information, see CSCee60212.

15.1.4 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.

15.1.5 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.

NTP expresses time in terms of UTC (Universal Coordinated Time), which has no notion of changing the time by an hour, such as changing time for summer time. `clock_get_time()` and other API functions call `check_for_summer_time()` to convert into summer time.

15.1.6 System Clock Changes

The system clock or wall clock time is changed in two ways:

- slew, which is a gradual change and is used, for example, by NTP for small corrections to the system clock.
- set, which relates to big time changes and is instantaneous. This is used, for example, by the CLI to set the time to any arbitrary value.

15.1.7 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.

15.2 Get the Current Time

Table 15-1 describes the Cisco IOS functions provided to get the current time from the system clock.

Table 15-1 Functions to Get the Current Time from the System Clock

Time to Get	Function
Current time at a medium resolution (roughly 4 ms)	<code>void clock_get_time(clock_epoch *epoch);</code>
Current time epoch expressed in microseconds	<code>ulong clock_get_microsecs(void);</code>
Time at the highest resolution supported by the hardware	<code>void clock_get_time_exact(clock_epoch *epoch);</code>
Current time in UNIX format	<code>ulong unix_time(void);</code>
Current time in terms of seconds and nanoseconds since 0000 UTC 1 January 1970	<code>void secs_and_nsecs_since_jan_1_1970(secs_and_nsecs *time);</code>
Current time in ICMP timestamp format (milliseconds since 0000 UTC today)	<code>ulong clock_icmp_time(void);</code>

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);
```

15.3 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);
```

15.4 Convert between Time Formats

Table 15-2 lists the functions available for converting between different time formats.

Table 15-2 Functions for Converting between Different Time Formats

Convert from...	Convert to ...	Function
<code>clock_epoch</code>	<code>ios_timeval</code>	<code>void clock_epoch_to_timeval(clock_epoch *epoch, ios_timeval *tv, long tz_offset);</code>
<code>clock_epoch</code>	UNIX time value	<code>ulong clock_epoch_to_unix_time(clock_epoch *epoch);</code>
<code>ios_timeval</code>	<code>clock_epoch</code>	<code>void clock_timeval_to_epoch(ios_timeval *tv, clock_epoch *epoch);</code>
<code>ios_timeval</code>	UNIX time value	<code>ulong clock_timeval_to_unix_time(ios_timeval *tv);</code>
UNIX time value	<code>clock_epoch</code>	<code>void unix_time_to_epoch(ulong unix_time, clock_epoch *epoch);</code>
UNIX time value	<code>ios_timeval</code>	<code>void unix_time_to_timeval(ulong unix_time, ios_timeval *tv, char **tz_name);</code>

15.5 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);
```

15.6 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);
```

15.7 Format Time Strings

The Cisco IOS provides a rich set of functions for formatting ASCII strings from epochs of various forms.

Format Time Strings

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 an `ios_timeval` with very flexible formatting options, use the `format_time()` function:

```
ulong format_time(char *buf, ulong buf_len, char *fmt, ios_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 `%CC` format descriptor:

```
printf("%CC", 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

16.1 Overview: Timer Services

Cisco IOS Timer Services support different types of timers. Each type can track time to the limits of the system clock accuracy, down to a granularity (for example, 4 milliseconds `sys_timestamp` layout). The following are the timer types:

- Passive timers. Passive timers note the current value of the timer variable `msclock` and then examine this value when triggered by another event. A single passive timer may be “watched” by the operating system; that is, when the timer expires, the process gets woken up. (Uses 4 bytes of system memory for one passive timer.)
- 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. All managed timers are “watched” by the operating system and a process is awoken when one of the managed timers expires. (Uses 32 bytes of system memory for each of the multiple managed timers. Does not scale.)
- Timer Wheel timers. Timer Wheel timers are used to provide a more efficient timer service when applications have scaling problems with managed timers. A Timer Wheel is a circular array of ordered timer “buckets” where the sorting has been done by the ordered array structure approach. Each bucket contains a list of timers that have the same expiration time. (Uses 12 bytes/16 bytes of system memory for each of the timer wheel timers. Does scale.)

Note Cisco IOS timer services questions can be directed to the `interest-os@cisco.com` alias.

The timer services are used for all time-related functions in the system, such as periodic processes, timeouts, and delay measurements. They are used in all cases where time *intervals* matter. Time-of-day is a separate function. It is discussed in the “Time-of-Day Services” chapter.

This chapter includes the following topics:

- Timer States
- Passive Timers
- Managed Timers
- Timer Wheel Timers
- Choose Which Type of Timer to Use
- Determine System Uptime

16.1.1 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.

16.1.2 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.

16.1.3 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.

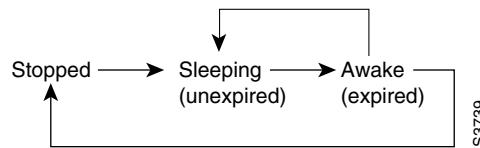
16.2 Timer States

Passive timers in the future and managed timers can be in one of three possible states (see Figure 16-1):

- 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 16-1 Timer States



33739

16.3 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*. In addition, a single passive timer in the future may be “watched” by the operating system, allowing the process to be awoken on expiration of that timer.

The data structure for a passive timer is a simple timestamp, `sys_timestamp`. This variable type is 32 bits in Release 11.0 and earlier, and is 64 bits in Release 11.1 and later.

16.3.1 Passive Timer Delay Descriptors

To support easy conversion from the Release 11.0 and earlier 32-bit timestamp and the Release 11.1 and later 64-bit time stamp, delay factors can be specified at 32-bit (`ulong`) format or 64-bit (`ulonglong`) format. Many operations have two similar functions, one supporting a delay using 32 bits, a second supporting a delay specified in 64 bits.

16.3.2 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.

16.3.2.1 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.
- 3 Optionally, one passive timer in the future may be “watched” by the operating system. When that timer expires, the process is automatically awoken.

16.3.2.2 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 jittered 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 days, it will suddenly look unexpired again. You should always stop or restart timers as appropriate after they expire.

16.3.2.3 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)
```

16.3.2.4 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)
```

16.3.2.5 Determine the State of Passive Timers in the Future

Table 16-1 lists the macros that allow you to determine the state of a passive timer in the future.

Table 16-1 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).	boolean <code>TIMER_RUNNING(sys_timestamp timer)</code>
Determine whether a timer is both running (that is, nonzero) and sleeping.	boolean <code>TIMER_RUNNING_AND_SLEEPING(sys_timestamp timer)</code>
Determine whether a timer is both running (that is, nonzero) and awake.	boolean <code>TIMER_RUNNING_AND_AWAKE(sys_timestamp timer)</code>
Determine whether a timer has expired.	boolean <code>AWAKE(sys_timestamp timer)</code> boolean <code>XAWAKE(sys_timestamp timer,</code> <code>ulong maximum_delay)</code>
Determine whether a timer has not yet expired.	boolean <code>SLEEPING(sys_timestamp timer)</code> boolean <code>XSLEEPING(sys_timestamp timer,</code> <code>ulong maximum_delay)</code>
Calculate time left sleeping before a timer expires.	ulong <code>TIME_LEFT_SLEEPING(sys_timestamp timer)</code> ulonglong <code>TIME_LEFT_SLEEPING64(sys_timestamp timer)</code>

16.3.2.6 Guidelines for Using the SLEEPING and AWAKE Macros in Releases Prior to Release 11.1

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.

16.3.2.7 Guidelines for Using the XSLEEPING and XAWAKE Macros in Releases Prior to Release 11.1

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 5 seconds 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 seconds, 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.

16.3.2.8 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.

16.3.2.9 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)
```

16.3.2.10 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)
```

16.3.2.11 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 time 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)

```

16.3.2.12 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 `timer`. 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)

```

16.3.2.13 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))
...

```

16.3.3 Watching a Single Passive Timer

As mentioned above, a single passive timer can be “watched” by the operating system, allowing the process to be awoken when that timer expires. The single watched passive timer uses the same functions as normal passive timers with the addition of the function [process_watch_timer\(\)](#).

16.3.3.1 Example: Watching a Single Passive Timer

The timer is established as indicated in the preceding section, then the process sleeps, waiting for that timer to expire. When the process is awoken, it performs similarly to when awoken by another event.

For more information on general event management, see Chapter 3, “Basic IOS Kernel Services.”

Initialization Routine

```
/*
 * Establish initial expiration time
 */
TIMER_START(snark_pdb->update_timer, SNARK_UPDATE);
/*
 * Notify the operating system that this is a watched timer
 */
process_watch_timer(&snark_pdb->update_timer, ENABLE);
```

Periodic Awaken Routine

```
while (TRUE) {
    process_wait_for_event()
    while (process_get_wakeup(&major,&minor)) {
        switch (major) {
            case TIMER_EVENT:
                snark_update()
                TIMER_UPDATE(snark_pdb->update_timer, SNARK_UPDATE);
                break;
            default:
                errmsg (&msgsym(UNEXPECTEDEVENT, SCHED), major, minor);
                break;
        } /* end switch */
    } /* end while process_get_wakeup */
} /* end while (TRUE) */
```

snark_block

```
if (AWAKE(snark_pdb->update_timer))
...
```

16.3.4 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 every 49 days—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.

16.3.4.1 Determine the Current Time

To obtain the number of milliseconds since system boot, 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)
```

16.3.4.2 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)
```

16.3.4.3 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)
```

16.3.4.4 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)
```

16.3.4.5 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);
}
```

16.3.5 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 days 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)
```

16.4 Managed Timers

16.4.1 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.

16.4.2 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.

16.4.3 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.

16.4.4 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.

16.4.4.1 Using Interrupt Routines and Managed Timers

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.

An interrupt routine can be used to bump the system clock. However, it will not force a process switch as IOS follows the RTC (Run To Completion) or cooperative multitasking model. When a process switch occurs, an expired timer may have made a process runnable or at least added to the work load of a process waiting to run. Note that the standard IOS tick frequency is 4 milliseconds (250 ticks per second).

16.4.4.2 Active, Stopped, and Expired Managed Timers

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. However, 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.

Under certain circumstances the scheduler may deliver a `TIMER_EVENT` to your process via the `process_get_wakeup()` call even if the managed timer that your process is watching has been stopped or is running but has not expired at the instant your process calls `process_get_wakeup()`. This happens when the scheduler sees that the timer has expired and records that fact by declaring a `TIMER_EVENT` pending for your process, yet something else (some other process or an interrupt handler, perhaps) stops the managed timer or modifies its expiration time before your process gets a chance to run and call `process_get_wakeup()`. If this is a possible scenario for any of the managed timers your process watches, then when handling a `TIMER_EVENT`, your process must be prepared to discover that the managed timer has already been stopped, or is running and sleeping (instead of running and expired), or that `mgd_timer_first_expired()` returns `NULL`; these are normal occurrences for such a process and are not errors.

16.4.5 `mgd_timer` Data Structure

The primary data structure for managed timers is of type `mgd_timer`. 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.

16.4.6 Guidelines for Using Managed Timers

You need 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.

16.4.7 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);
```

16.4.8 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);
```

16.4.9 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);
```

16.4.10 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 (parent timers can only have context through additional context, see the `mgd_timer_set_additional_context()` function under “16.4.16.2 Set Extended Context” for more information).

```
void *mgd_timer_context(mgd_timer *timer);
```

16.4.11 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);
```

16.4.12 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);
```

16.4.13 Set a Leaf Timer’s Expiration

To change a leaf timer’s expiration to the value of a timestamp if that value is sooner than 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);
```

16.4.14 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 complaints that it is an uninitialized timer or cause crashes with managed timer APIs, such as `mgd_timer_set_exptime_internal()` or `mgd_timer_walk_down_tree()`, that walk through the tree in the backtrace.

16.4.15 Determine the State of a Managed Timer

Table 16-2 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 16-2 Functions to Determine the State of Managed Timers

Description	Function
Determine whether a timer is running.	<code>boolean mgd_timer_running(mgd_timer *timer)</code>
Determine whether a timer is both running and sleeping; that is, whether it is unexpired.	<code>boolean mgd_timer_running_and_sleeping(mgd_timer *timer)</code>
Determine whether a timer is both running and awake; that is, whether it is expired.	<code>boolean mgd_timer_expired(mgd_timer *timer)</code>
Return the address of the first expired timer in the timer tree.	<code>leaf_timer = mgd_timer *mgd_timer_first_expired (mgd_timer *parent_timer)</code>
Return the address of the first running timer in the timer tree.	<code>leaf_timer = mgd_timer *mgd_timer_first_running (mgd_timer *parent_timer)</code>
Return the number of milliseconds left before a timer expires.	<code>long mgd_timer_left_sleeping(mgd_timer *timer); longlong mgd_timer_left_sleeping64(mgd_timer *timer)</code>
Return the expiration timestamp for a timer.	<code>sys_timestamp mgd_timer_exp_time(mgd_timer *timer)</code>

16.4.16 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.

16.4.16.1 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);
```

16.4.16.2 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);
```

16.4.16.3 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);
```

16.4.16.4 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);
```

16.4.16.5 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.

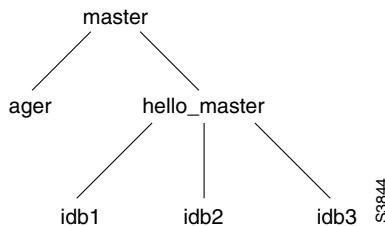
```
mgd_timer *mgd_timer_first_child(mgd_timer *parent_timer);
```

16.4.17 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 `master`, under which is the ager timer `ager` and a parent timer `hello_master`. Under `hello_master` there are individual hello timers for each interface. Figure 16-2 illustrates this structure.

Figure 16-2 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);

```

16.5 Timer Wheel Timers

A timer wheel is a fixed size list of queues which is processed as a circular queue, hence the term “wheel”. Each tick is a queue in the timer wheel holding events that are to occur at a given time. The elapsed time between ticks is a constant for each timer wheel. The number of ticks in a timer wheel is fixed for any single timer wheel. Events are added to an appropriate queue based on when the event is to occur relative to the entry that is current when the event is queued. When the elapsed time for a tick occurs, the next queue is made current and all events on that queue are processed.

The Cisco IOS Timer Wheel Timer model is a set of library API functions that are used when applications require the scalability that managed timers do not provide. The primary timer wheel structures must be embedded with each application. The generic timer wheel library does not cache any shared data structure.

For the purpose of memory optimization, there is not any additional context ID defined in the Timer Wheel Timer base or extended timer structure. If any application needs to associate a context ID with a timer, then it can wrap the timer structure within its context structure, and the context ID can be derived from the timer entry address without any extra overhead.

The timeout expiration is handled by the application’s process. It is possible to extend the timeout processing to a multiple priority service. With a multiple priority service, some types of time expiration can be handled immediately by the tick event handling process, and other types of time expiration can be deferred or passed to a separate process that handles the timeout event via the event manager. However, do not create a separate process to handle the time expiration or to generate a timeout event until you are sure that having the process will be an improvement that outweighs the extra context switching overhead that having the process may add.

16.5.1 Timer Wheel Terms

Timer Wheel

A timer wheel is a nearly ordered list of timers where the sorting is done with a bucket approach. The time to wait is divided by the granularity of the buckets to find the bucket to link the timer into. Each node in the list indicates a slot of a timer wheel. It is also called a time bucket. Each bucket contains a list of timers that have the same expiration time.

Timer Wheel Granularity

The time value between any two adjacent buckets is the granularity (or one tick). The amount of time between which we process each bucket. If the granularity is large, events bunch together and the timing is less precise. If the granularity is small, much overhead is spent starting up the timer expiration process and more memory is needed for the buckets.

16.5.2 Timer Wheel Timers Background

The IOS core generic timer model is an enhanced version of the timer wheel model implemented originally for PPP. The IOS PPP component has implemented the timer wheel model with multiple levels. This model allows the timer wheel timer to support a relatively large number of durations with a limited timer wheel array size. It maps the time delay value into an appropriate timer wheel bucket. Whenever the lower level wheel completes a cycle, it moves the timer from its next upper level wheel down to a corresponding bucket in the lower level of the wheel. However, this model adds some additional overhead in moving data from the upper level of the wheel down to the lower level of the wheel. This section describes how this model works and then discusses the profiling results regarding overhead and discusses how the model can be optimized.

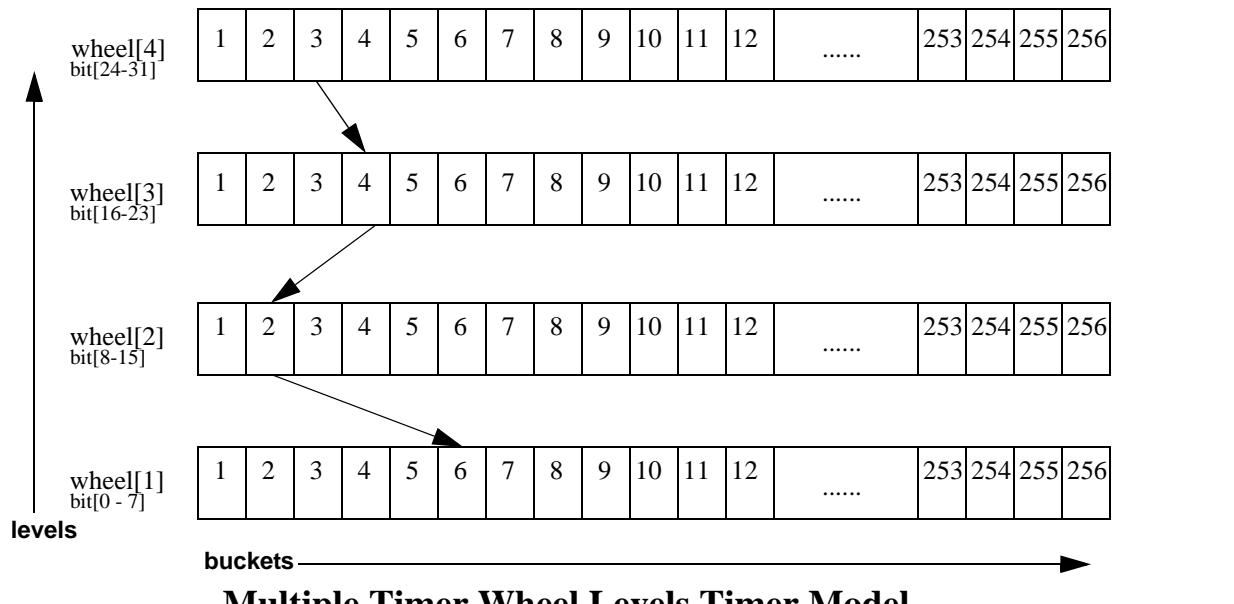
Figure 16-3 shows a typical multiple timer wheel model using a 32-bit timer. Each level of the wheel has a different granularity. The interval between two adjacent buckets of the lowest level (level 1) of the timer wheel (wheel[1]) is the granularity value, the interval between the buckets of wheel[n] is $2^{(n-1)} * \text{granularity}$, where n indicates the level of the wheel. Each timer caches the time tick value so that the model uses the value to decide the location of the timer to be inserted. The lowest 8 bits of the current tick value points to the bucket whose timers expire currently.

Figure 16-3 Multiple Timer Wheel Levels Timer Modelwheel size = $2^8 = 256$

number of wheel levels = 4

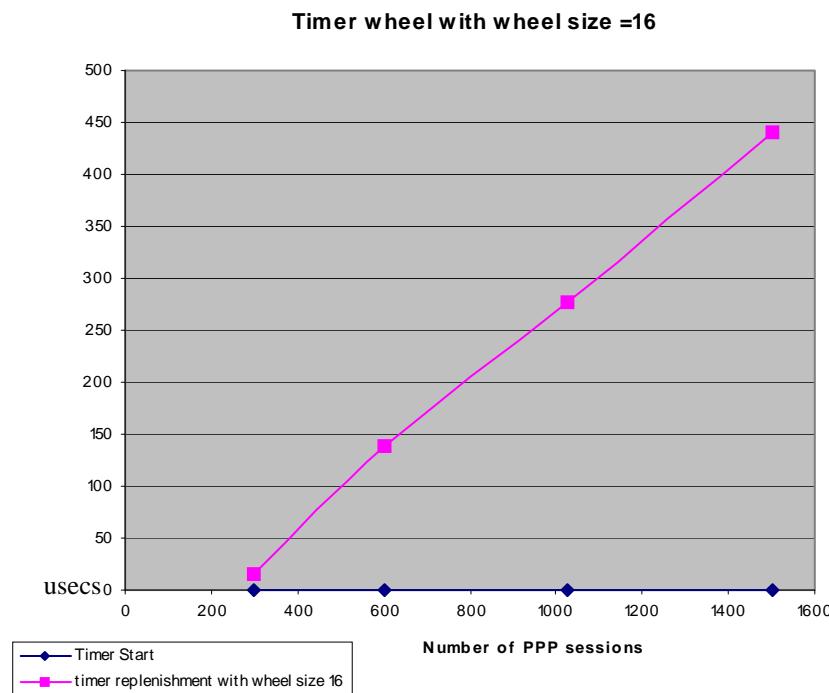
level 1 granularity = 16 msec

max time duration = 48 days

**Multiple Timer Wheel Levels Timer Model**

From the model algorithm point of view, the data moving operation (or timer replenishment) only happens when a wheel completes a cycle and the bucket is not empty. If there is no data to move, its overhead is O(1). Assuming w_s is the wheel size, for each w_s time tick, the timer model will need to move the timers from an upper level of the wheel into the next lower level of the wheel (for example, from wheel level 4 to wheel level 3). This data moving overhead depends on the number of timers in a bucket and the w_s value. If w_s is larger, then most or all the timers can be mapped into the lowest level of the wheel, wheel [1], and the interval of two data moving operations is quite large. In other words, this operation is not significant because it isn't triggered very frequently. This frequency is determined by both the w_s and the granularity parameters. By adjusting those values, the overall CPU efficiency of the timer wheel model can be further optimized.

The PPP timer wheel component was profiled under a different number of PPP sessions. Figure 16-4 shows that the overhead of data moving varies with the number of PPP sessions under a wheel size of 16 (the current PPP timer wheel size). The average elapsed time (usec) of the timer insertion is a constant value (0.6 usec), and it scales well and does not change when the number of PPP sessions increases. However, the individual timer replenishment overhead increases when the number of outstanding timers in a bucket increases. In the test case, there is no data traffic and no client application. The maximum time duration is 2000 msec. For a granularity of 32 msec and wheel size of 16, most of the timers are mapped into wheel level 2 and wheel level 4. During the flapping time, each individual timer replenishment overhead is not small. Figure 16-4 shows that the average elapsed time (usec) of the timer replenishment operation and the timer insertion varies depending on the number of PPP sessions.

Figure 16-4 Timer Wheel with Wheel Size = 16

When the wheel size is increased to 256, all the timers are inserted into the lowest level of the wheel. Figure 16-5 shows that the profiling data for the average elapsed time (usec) for timer replenishment is reduced to $O(1)$ since there is no data needed to be moved.

If the wheel size is set to 128, it will also give the same results, since all the timers are inserted into the lowest level of the timer wheel. There is no data that is needed to be moved. The timer insertion overhead is $O(1)$ and is unchanged. It uses less memory for the timer wheel structure.

Figure 16-5 Timer Wheel with Wheel Size = 256
Timer Wheel with wheel size = 256

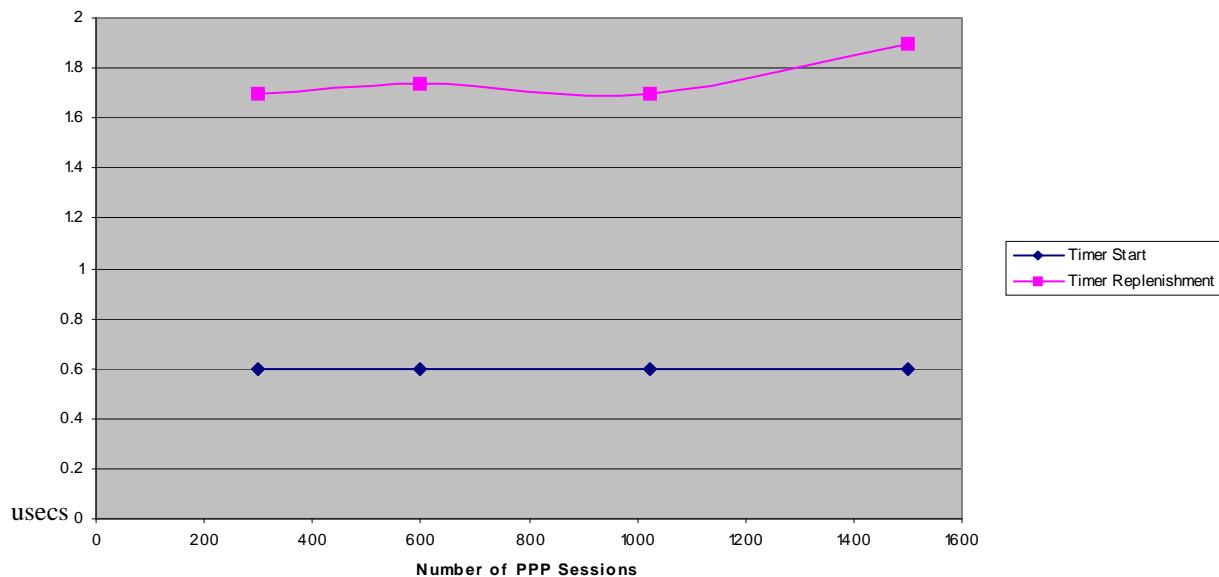
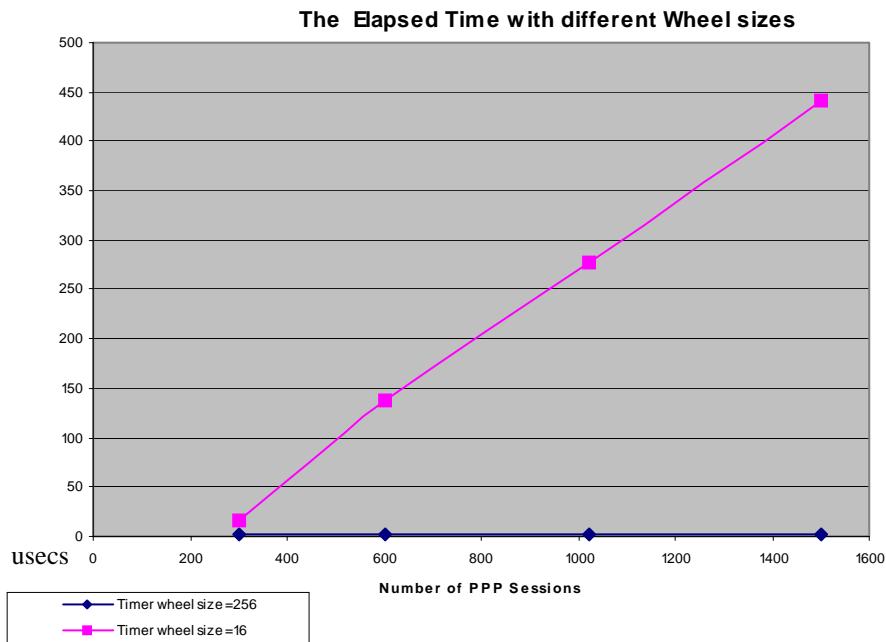


Figure 16-6 shows the comparison of the average data moving elapsed time (usec) between different wheel sizes.

Figure 16-6 The Elapsed Time with Different Wheel Sizes



The profiling results supported the prediction that the data moving overhead of the multiple timer wheel levels model is adjustable to achieve better system efficiency. Therefore, this model was added to Cisco IOS.

16.5.3 Timer Wheel Timers Benefits

The following types of applications benefit from using the timer wheel timer model:

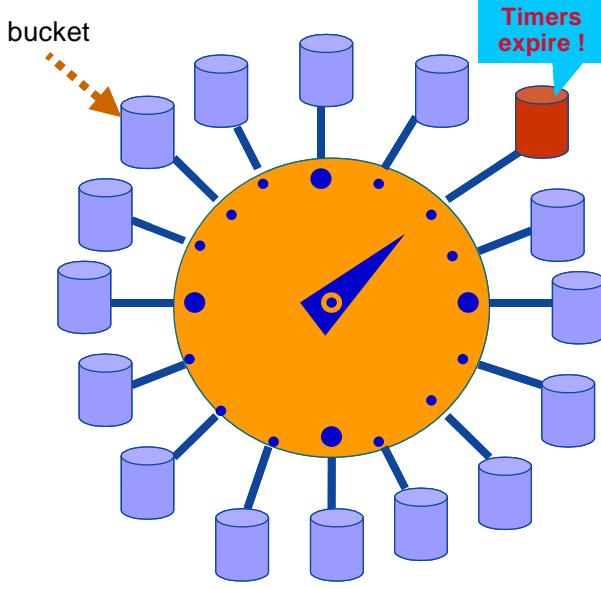
- Applications that have a larger number of timers.
- Applications that frequently start or restart the timer.
- Most applications that have timers that are stopped before they expire.

It does not matter whether an application's timers have an equal time duration or not, the timer wheel always provides a fast way to start and stop a timer. With the managed timer, only those applications that can always group the timers with an equal timer delay value into the hierarchical tree can have the optimized timer insertion overhead at $O(\log N)$. Unfortunately, not all applications can use this optimization because if the time duration value is a variable, then the application can have a scaling problem with the managed timer in a large configuration environment.

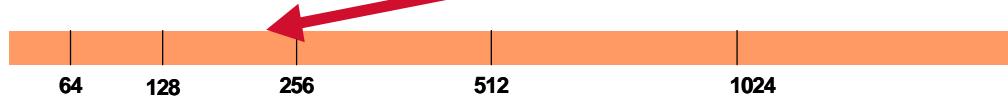
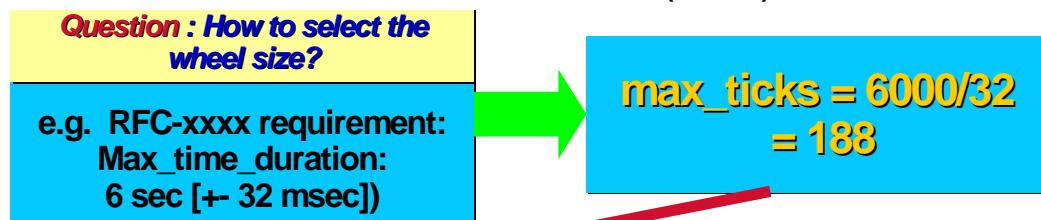
The Multiple Timer Wheel Levels Timer Model has an additional data moving cost because it needs to move the timer from a high wheel level to the next lower wheel level during the tick process time. However, in general, the data moving operation is only triggered every w_s tick time interval. If the timer bucket is empty, it does nothing and the overhead is $O(1)$. This overall overhead is still acceptable. In addition, if the w_s wheel size is adjusted, then the data moving overhead can be reduced to $O(1)$ or the data moving operation is rarely involved.

The single timer wheel timer model provides an efficient timer service for the application that has a small time duration value. The single timer wheel model is a special case of the multiple timer wheel. If an application's maximum time duration is known, then the number of wheels can be adjusted to minimize the wheel structure size. For example, if an application can map all its timers into the lowest wheel level, then a single wheel will work. Figure 16-7 shows an example of a single timer wheel timer.

Figure 16-7 Single Timer Wheel Timer



- The time is evenly divided by the granularity.
- Each bucket indicates a specific number of time ticks.
- The time value between any two adjacent buckets is the granularity (or one tick).
- The time tick event rolls the timer wheel.
- The time tick hand points at the bucket whose timers currently expire.
- Max time duration is **wheel size x granularity** (msecs)



**Recommended wheel size will be
256 (buckets)**

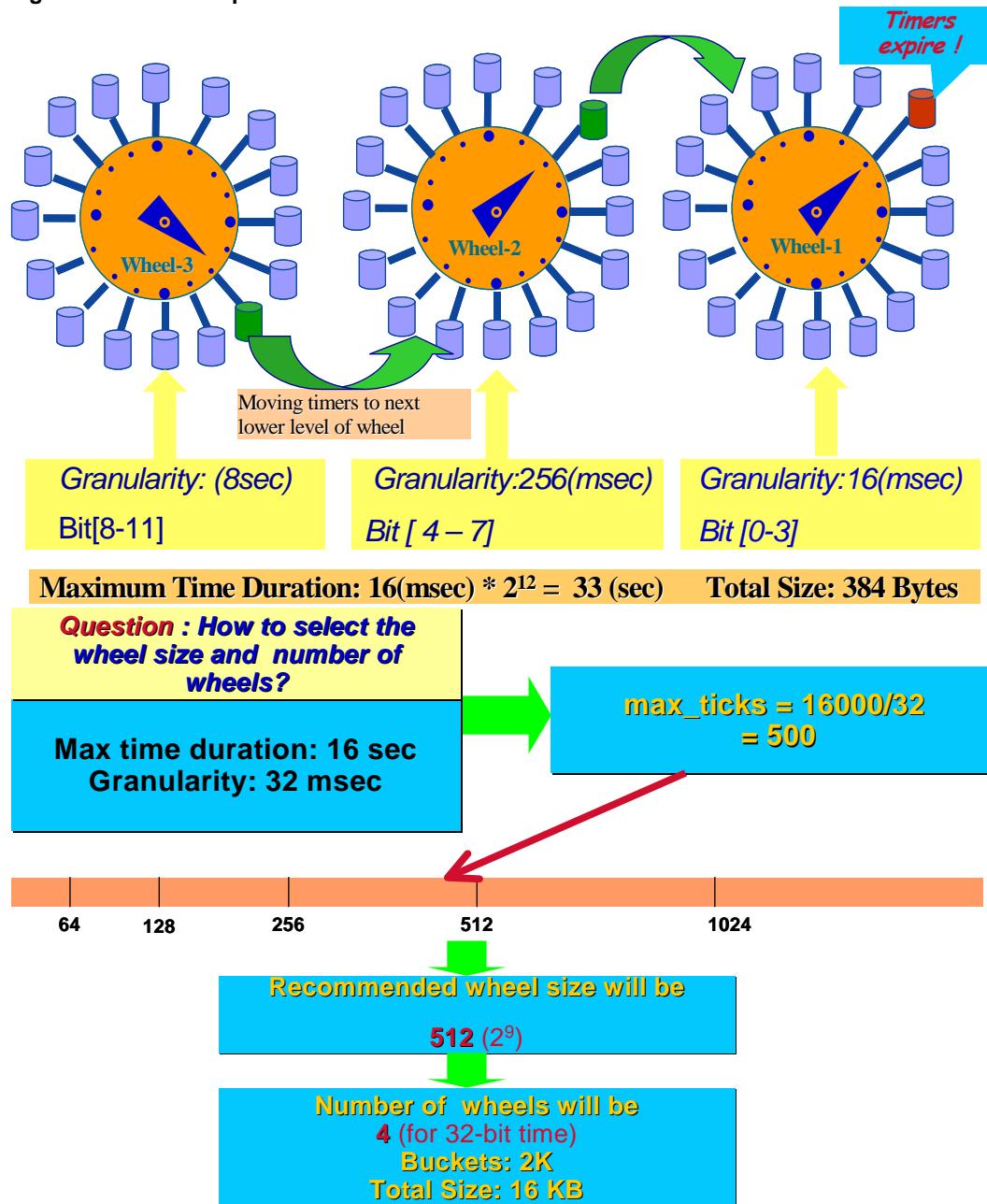
It is possible to insert a large number of timers into the same bucket (the timers will expire at the same time) because the timer bucket does not maintain the ordering in the same bucket with the timers. By using a jittered time delay, the timers can be spread among a number of buckets. This helps avoid the problem of too many timers being inserted into the same timer bucket, which could potentially cause a CPU hog during the time expiration processing.

The Multiple Timer Wheel Levels Timer Model supports 4-bit timers up to 32-bit timers with a limited wheel structure, `tw_timer()`. By adjusting the timer wheel size and the wheel levels you can further optimize the timer wheel timer model for each application.

From the memory usage point of view, the timer wheel model helps to save more than 50% of the memory space required for a managed timer in a large configuration environment. The `tw_timer` structure consists of 13 bytes of data that contains the information necessary for a timer wheel to perform the timer service. The current `mgd_timer` structure consists of 32 bytes of data. For a system that has 100k timers, the `mgd_timer` needs 3.2 MB of memory space, and the `tw_timer`

only needs 1.3 MB of memory space. Therefore, the timer wheel timer model saves 1.9 MB of the system's memory. This is a significant memory usage optimization and makes the system more scalable. Figure 16-8 shows an example of a multiple timer wheel timer.

Figure 16-8 Multiple Timer Wheel Timer



16.5.4 A Generic Timer Wheel Timer Model

Since applications other than PPP also face scalability issues when using managed timers, a generic timer wheel model library is needed for those applications to avoid duplications of effort for each individual application. Therefore, the multiple timer wheel timer for PPP was moved into the IOS infrastructure and enhanced as a generic timer wheel model. This model is implemented as a set of

library API functions. Each component's process still needs to handle the time tick event and pass it to the timer wheel API to handle the time expiration and to perform other timer wheel operations.

Since the timer wheel timer API is a set of library routines, there are no shared data structures. The primary data structures is completely opaque. Users should not directly access either of the two structures, `tw_wheel` and `tw_timer` or `tw_timer_ext`. Each application should follow the procedures defined in `tw_timer.h`.

There is an extended timer structure called `tw_timer_ext`. This structure is defined to support the Timer Wheel Timer facility's extended mode. When the granular value is less than an application's maximum timer types, the application must set the Timer Wheel Timer mode to extended mode and use `tw_timer_ext` as the timer structure.

16.5.4.1 Primary Timer Wheel Data Structures

The generic timer wheel module is composed of three primary data structures:

- 1 The `tw_wheel` structure holds the common parameters and the timer wheel array for holding the timers.
- 2 The `tw_timer` is the base timer structure, and it has the dual-link to the timer queue bucket if the timer is enabled. The `tw_timer` also has the time tick and timer type used by the timer wheel. The timer type is saved in the lower granularity bits. This base timer structure supports the timer wheel base mode. The base mode is available only for applications that have a maximum number of timer types that is not greater than the required granularity value.
- 3 The `tw_timer_ext` structure is the extended timer structure, and it has an additional byte for the timer type. This extended timer wheel mode is used for those applications that have a maximum timer types value greater than the required granularity value.

16.5.5 Timer Wheel Timers API

This section lists the generic Timer Wheel Timers API functions that can be used by other applications to create their own Timer Wheel Timer service. The following Timer Wheel Timers API functions are defined in the `sys/h/tw_timer.h` header file:

- 1 `tw_timer_get_type()`
- 2 `tw_timer_get_type_ext()`
- 3 `tw_timer_granularity()`
- 4 `tw_timer_init_wheel()`
- 5 `tw_timer_init_wheel_ext()`
- 6 `tw_timer_load_tbl()`
- 7 `tw_timer_remaining()`
- 8 `tw_timer_running()`
- 9 `tw_timer_set_handler()`
- 10 `tw_timer_set_type()`
- 11 `tw_timer_set_type_ext()`
- 12 `tw_timer_start()`
- 13 `tw_timer_start_jittered()`
- 14 `tw_timer_stop()`
- 15 `tw_timer_tick()`

See the *Cisco IOS API Reference* for more information on each of the Timer Wheel Timers API functions.

16.5.5.1 Create and Initialize The Timer Wheel API

The timer wheel must be initialized before any timer can be started. Either the `tw_timer_init_wheel()` or the `tw_timer_init_wheel_ext()` function must be invoked during subsystem initialization time. The `base_mode` indicates whether the timer wheel is using the base timer structure `tw_timer` or the extended timer structure `tw_timer_ext`. An application can keep its own timer type mapping table by setting the `handler` field to its timeout handle. It can also let the timer wheel keep the timer type mapping table for it by setting the `handler` field to `NULL`. The application must load the timer table into the timer wheel after the wheel is initialized if it wants the timer wheel to handle the timer type mapping. The timer wheel supports interrupt safety capability. When the interrupt safe flag is set to `TRUE`, the timer wheel will enter the interrupt safety mode. If the timer wheel can be started or stopped from an interrupt level handler, then the interrupt safe flag must be set to `TRUE` during the wheel initialization process.

16.5.6 Steps To Create A Timer Wheel Timer Service

Complete the following steps to create a Timer Wheel Timer service.

Step 1 Define your timer structures.

The two primary timer wheel structures must be embedded with the application's structures. For example:

```
tw_wheel my_wheel;
structure my_context_ {
...
tw_timer timer_a;
tw_timer timer_b;
...
};
/* Define the timer type */
enum my_timer_type {
TIMER_A = 1,
TIMER_B };
```

Step 2 Define your timer handler table.

An application can handle its own timeout handler mapping by setting the `handler` field to its common timeout handler during the timer wheel initialization. It can also let the timer wheel model handle the mapping by setting the `handler` field to `NULL` and loading the mapping table after the timer wheel is initialized.

- The following sample code illustrates the first option, which uses the *handler* field to let an application define its own timeout handler mapping:

```

/*
 * Define PPP timer wheel structure
 */
tw_wheel ppp_timer_wheel;

/*
 * PPP timer wheel timer expiration handler lookup table.
 */
tw_timer_handler ppp_timer_handler[PPP_TIMER_MAX] = {
    NULL,
    ppp_fsm_timeout,
    ppp_negotiation_timeout,
    ppp_authentication_timeout,
    ppp_idle_timeout,
    ppp_periodic,
    ppp_lqm_timeout,
    ppp_mcb_timeout,
    ppp_bacp_timeout,
};

/*
 * ppp_timeout_handler
 * Invoked by the timer wheel. It maps a timer type into
 * the timer handler and triggers the individual timer
 * expiration processing.
 */
void ppp_timeout_handler (tw_timer *tmr)
{
    tw_timer_type type;

    type = tw_timer_get_type(&ppp_timer_wheel, tmr);
    if ((type < PPP_TIMER_MAX) && ppp_timer_handler[type]) {
        (*ppp_timer_handler[type])(tmr);
    }
}

```

- The following sample code illustrates the second option, which uses the *handler* field set to NULL to let the timer wheel model define the timeout handler mapping:

```

/*
 * Timer handlers
 */
extern void a_handler (tw_timer *tmr);
extern void b_handler (tw_timer *tmr);
/*
 * Define the timer table
 */
tw_timer_table my_timer_table = {
{TIMER_A, a_handler},
{TIMER_B, b_handler},
{0, NULL}};

```

Step 3 Initialize your timer wheel.

The timer wheel must be initialized before any other timer can be started. The `tw_timer_init_wheel()` function or the `tw_timer_init_wheel_ext()` function must be invoked in the component subsystem initializing routine.

- The following sample code illustrates the first option, where an application defines its own timeout handler mapping:

```
/*
 * Create and initialize the Timer Wheel structure which
 * supports the timer wheel timer service.
 * Ignore any possible failure response from the tw_timer
 * call as this should be caught by regression testing...
 */
(void)tw_timer_init_wheel(&ppp_timer_wheel,
                           TW_TIMER_DEF_GRANULARITY,
                           PPP_MAXIMUM_TIMER_DURATION,
                           PPP_TIMER_MAX,
                           TRUE,
                           FALSE,
                           ppp_timeout_handler);
}
```

- The following sample code illustrates the second option, where the timer wheel model defines the timeout handler mapping:

```
my_subsys_init ()
{
    ...
/*
 * Initialize the timer wheel
 */
tw_timer_init_wheel (&my_wheel,
                     TW_TIMER_DEF_GRANULARITY,
                     max_duration,
                     max_type,
                     TRUE,
                     FALSE,
                     NULL);
/*
 * Load the timer table
 */
tw_timer_load_tbl (&my_wheel, &my_timer_tbl);
}
```

Step 4 Set your time tick handle to passive timer.

The application needs to start the passive timer with the granularity value to generate the tick event, and to set the time event handler.

The `tw_timer_tick` function must be invoked within your timer event handle. For example:

```
static sys_timestamp my_tick_timer;      /* Periodic Timer Wheel timer */

my_time_event_handle () {
    TIMER_START(my_tick_timer, TW_TIMER_DEF_GRANULARITY);
    tw_timer_tick (&my_wheel);
}

my_application_process ()
{
    ...
    TIMER_START(my_tick_timer, TW_TIMER_DEF_GRANULARITY);
    process_watch_timer(&my_tick_timer, ENABLE);
    while (TRUE) {
        process_wait_for_event();
        while (process_get_wakeup(&major, &minor)) {
            switch (major) {
                case TIMER_EVENT:
                    my_time_event_handle ();
                    break;
                ...
                default:
                ..
            }
        }
    }
}
```

Then, the application is ready to start its timer.

16.5.7 Enhanced Timer Wheel (*New in 12.4T*)

The timer wheel timer facility is designed to support timers that can vary from two seconds (or less) up to a day. The short range timers are expected to be started and stopped (as responses are received) and not expired. In such cases, the timer wheel approach is preferred since a timer can be started and stopped with very little effort. The long range timers are expected to exist for the life of the connection. Some of them might have expired and restarted, but the period of time that the long range timers are expired might be relatively long compared to the short timers.

With simple timer wheel implementations (non-hierarchical), long-term timers are processed (but not expired) once per each complete turn of the timer wheel. This can result in significant extra overhead. Hierarchical timer wheels use multiple wheels to reduce this overhead. A timer is placed on the appropriate timer wheel when it is started. As each lower-level (faster) wheel completes a circuit, it takes one bucket from the next highest wheel and redistributes its timers onto the lower-level wheel. The timer does not change, only the linked list to which it belongs.

In the standard implementation of timer wheels, it is expected that the process using the timer wheel starts a passive timer with the delay equal to the “granularity” of the timer wheel and the process starts watching that passive timer. The scheduler doesn't know about timer wheels. The scheduler is only aware of the started passive timer and thus the scheduler wakes up the process every unit of granularity. The process (after waking) is expected to call the timer wheel internal routine, `tw_timer_tick()`. The `tw_timer_tick()` routine processes the tick event and rolls the wheel. If any timer has expired, `tw_timer_tick()` calls the associated handler to handle the expiry.

With the standard implementation, problems occur when an application has very fine granularity. For example, the granularity of timer wheels is 4 milliseconds in SAA. Therefore, the `saaEventProcessor` process is woken up every 4 milliseconds by the scheduler. Because of this, the `saaEventProcessor` process shows very high CPU usage. (See DDTs CSCec79807: High CPU 96% seen in IOU images with SAA/PPP.)

The enhanced timer wheels solve the problem of high CPU usage by providing to timer wheel users the option to push the responsibility of tracking timer expiration (in a timer wheel) from the process using the timer wheel to the scheduler. To provide backward compatibility with existing timer wheel users, this facility itself is optional and can be turned on when a timer wheel is created. When this option is turned on, a process using that timer wheel is woken up only when a timer has expired in the timer wheel.

16.5.7.1 Enhanced Timer Wheel Implementation

To allow the scheduler to track timer expiry in a wheel, the scheduler must become *aware* of the timer wheel. Since the scheduler knows only managed timers, it was decided to create a mapping between the timer wheel and the managed timer tree that the scheduler is aware of. To create this mapping:

- 1 A global managed timer named `tw_root_timer` is introduced and the scheduler's root timer, `sched_master_timer`, is made its parent. The type of the `tw_root_timer` timer is `STT_TIMER_WHEEL` and the `tw_root_timer` timer is fenced.
- 2 Then, every timer wheel struct has a new `tw_sched_linkage` member. `tw_sched_linkage` is a structure of type `tw_sched_links`, which among other things has a managed timer named `gran_timer` as its member.
- 3 A `watched_tmr_wheel` pointer to a timer wheel is maintained in the current active event set of the process using the timer wheel. The `process_watch_timer_wheel()` Enhanced Timer Wheel API function is provided to allow this `watched_tmr_wheel` pointer to point to the timer wheel that the process uses.
- 4 When the process using the timer wheel then calls `process_wait_for_event()`, the scheduler accesses the concerned timer wheel through the `watched_tmr_wheel` pointer and completes the following tasks:
 - (a) Starts the `gran_timer` managed timer with an expiry time equal to the granularity of the concerned timer wheel timer.
 - (b) Links this `gran_timer` managed timer to the `tw_root_timer` global managed timer, which completes the linkage to the scheduler's timer tree.
- 5 When `tw_root_timer` expires, the scheduler calls a newly introduced `process_wake_timer_wheels()` function. This function rotates each wheel whose timer has expired and if there are timers in that bucket, it wakes up the corresponding process with a `TIMER_WHEEL_EVENT` notification.

16.5.7.2 Enhanced Timer Wheel API

The Enhanced Timer Wheel facility provides its own interface to processes that use timer wheels. Therefore, existing code that uses timer wheels can continue to exist without any major changes. However, if you want to exercise the option of rotating the timer wheel from the scheduler, you have to use the Enhanced Timer Wheel API. The usage and semantics of the Enhanced Timer Wheel API

closely matches the IOS API's that allow a process to be notified for events like QUEUE or BOOLEAN. For example, SAA was modified to make use of the Enhanced Timer Wheel API functions. Here are the Enhanced Timer Wheel API functions:

1 `process_watch_timer_wheel(tw_wheel *tmr, boolean enable)`

This function can be called by a process to either watch a timer wheel or to disable watching a timer wheel. A process begins watching the timer wheel when the argument `enable` is TRUE and stops watching it when the argument `enable` is FALSE. A process can only watch one timer wheel at a time. When a timer present in the timer wheel expires, the process is woken up with a TIMER_WHEEL_EVENT event. The handling semantics are the same as for any other IOS events like QUEUE_EVENT events. For more information, see the [process_watch_timer_wheel\(\)](#) API reference page.

2 `tw_process_expiry (tw_wheel * tmr)`

This function is called by a process that is watching a timer wheel and that has been woken up because of a TIMER_WHEEL_EVENT notification. The notification of the timer wheel event simply wakes the process. Since the handlers registered with the timer wheel have to be called from the process context, process coders should call this API function when they get the TIMER_WHEEL_EVENT notification. For more information, see the [tw_process_expiry\(\)](#) API reference page.

16.6 Choose Which Type of Timer to Use

Follow these guidelines when deciding which type of timer to use:

- Use passive timers for items that are updated at data-forwarding time, because the overhead of a `mgd_timer_start()` function can be significant.
- 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.
- Both managed timers and timer wheel timers support multiple timers per process. Applications that have scaling problems with the managed timer can use the Timer Wheels Timer API to improve the system's efficiency. If there are more than 500 timer instances then the timer wheel provides a more efficient solution than the managed timer, in memory usage as well as CPU time.

Note that when using timer wheel timers, the application needs to be aware that the timers might not be fired in the order of their time value when the time offset value is within the granularity range. For example, assume that `t1` and `t2` are two timers, and that `t1` has the time value `td_val1` and `t2` has the time value `td_val2`. If $(td_val2 - td_val1) < \text{granularity}$, then both `t1` and `t2` can be inserted into the same bucket. Timers are not sorted in a bucket, so the `t2` timer can be fired before the `t1` timer even though time value `td_val2` is greater than time value `td_val1`.

16.6.1 The Effect of Changing a Watched Timer While Waiting for Events

When a process is watching a passive timer, the scheduler updates its data structures with the wakeup time that the passive timer has been set to only at the instant that the process calls `process_wait_for_event()`. Any changes made to the passive timer that the process is watching while the process is suspended in `process_wait_for_event()` have no effect on when or whether the process will wake up; this is because the changes do not propagate to any scheduler timer data structure (for example, see the code that handles the PROC_EVR_IDLE case in `process_idle_internal()` on `flo_isp`).

In contrast, changes made to watched managed timers while the watching process is suspended in `process_wait_for_event()` are propagated immediately to the scheduler timer data structures and affect the scheduling of the process. This means that if you want to have one process change the wakeup time of a second process or cause a second process to wake up by setting a timer that the second process is watching, then the second process cannot be watching a passive timer. However, a watched managed timer would work in this case.

16.7 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.

Note Cisco IOS strings and character output development questions can be directed to the format-scrub@cisco.com and os-infra-team@cisco.com aliases.

See also Chapter 11, “Standard Libraries”, for information about the POSIX and ANSI C standard library functions, including standard string functions, supported by the Cisco IOS software.

See also Chapter 20, “Debugging and Error Logging” for information about gaining UTF character support via the TCL package.

17.1 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.

17.1.1 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.

17.1.2 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, ...);
```

17.1.2.1 Differences between buginf() and printf()

`buginf()` should be used only in conjunction with an enabled debug flag that can be turned on/off via debug CLI. `printf()` should not be used in the switching path. Furthermore, `printf()` cannot be used when interrupts are disabled.

The significance of `\n` as the first character in the format string is particular to `buginf()`.

`buginf()` writes its message into a buffer that is output later. The write operation is atomic, and therefore is interrupt-safe. `printf()` uses assorted buffers in between the API and the actual hardware. Its big problem is that it can *block* if those buffers get full. Also `printf()` uses `forkx->tty`, so if used in interrupt context (even if that would work), it can write to random terminals/vtys/etc. `printf()` should never be used in a critical section since it may relinquish the process it is running in.

`buginf()` is output using the logger code. `buginf()` output can be directed to any of the same destinations that log output goes to. `printf()` goes to the process's tty (or to the console if the process has no tty); there is no other choice. `printf()` should never be used except in response to user input, or for certain boot-time messages that must be output to the console before the logger is running.

From a usability point of view, `printf()` is more commonly used in response to user input (for example, in `exec()` task) and `buginf()` is used to output diagnostic messages.

In a parser action routine, you must not send a message to another task and have that task generate output using `printf()`. While this appears to work when using the console, it does not work when using a telnet port.

17.1.3 Print a String into a Buffer

To format strings and place them into a buffer, use the `snprintf()` function (this function is typically safer to use than the `sprintf()` function). The formatting strings for this function are the same as those for the `printf()` function.

```
int snprintf (char *s, size_t n, const char *format, ...);
```

17.1.4 Print a String to Nonvolatile Storage

To write configuration strings to nonvolatile storage, call the `nv_write()` function.

```
void nv_write(boolean predicate, char *string, ...);
```

17.1.5 Turn on Automatic "---MORE---" Processing

To turn on automatic “more” (*automore*) processing, call the `automore_enable()` function.

```
void automore_enable (const char *header)
```

After calling this function, `printf()` will pause after displaying each page, asking the user if they want more by displaying a prompt of the form “---MORE---”. The automore processing resulting from the user’s response is as in Table 17-1:

Table 17-1 automore Input Prompt Processing and Results

automore Input at Prompt	automore Result
? (question mark)	Display a help string on acceptable responses to the automore prompt.
Space, 'Y', or 'y'	Do automore page processing: display one more page of output.
'\r' (return)	Do automore line processing: display one more line of output.
Any other character	Abort the automore session: stop displaying any more output lines.

The *header* parameter specifies a header string to display at the beginning of each page for this automore processing session. Use the `automore_header()` function if you want to change the header displayed for each automore page of output during the session (see section 17.1.6, “Change Automore’s Header in Midstream”).

The page size is the terminal length setting (**term len**). If the terminal length setting is zero (**term len 0**), enabling automore processing has no effect. Use the `automore_conditional()` function to conditionally control automore processing when there are a particular number of lines left in relation to the terminal length setting. See section 17.1.9, “Conditionally Prompt to Do More Output”, for details.

If the automore session is aborted, further output will be suppressed until automore processing is disabled by calling the `automore_disable()` function (see section 17.1.7, “Disable “---MORE---” Processing”).

Note Routines must call `automore_disable()` before exiting.

17.1.6 Change Automore’s Header in Midstream

Once automore processing is running, it is sometimes useful to be able to change the header displayed with each page of output. Use the `automore_header()` function to specify a different header string midstream:

```
void automore_header (const char *header)
```

17.1.7 Disable “---MORE---” Processing

To disable automore “---MORE---” processing, call the `automore_disable()` function.

```
void automore_disable(void)
```

17.1.8 Find Out if User has Quit Automore

To find out whether the user has quit out of automore processing, call the `automore_quit()` function.

```
boolean automore_quit (void)
```

Use this function to discover (usually by polling the return value) that the user indicated to quit automore processing, and take appropriate action. If the function returns TRUE, then the output is being flushed; if FALSE, output is not being flushed.

17.1.9 Conditionally Prompt to Do More Output

To conditionally prompt to do more output, call the `automore_conditional()` function.

```
void automore_conditional(int lines)
```

If this function is called with a value of 0 or a value greater than the page size, the automore “---MORE---” prompt is displayed at the next output, and the output pauses at that point.

If `lines` is greater than 0 and less than the page size, automore processing displays the “---MORE---” prompt at that point if there are `lines` or less lines left in the page to display. In other words, if (number of lines already displayed in the page) + `lines` \geq (page size), display the automore prompt and pause the output.

For example, consider a case where `automore_conditional()` is called with `lines`=5, the page size is 23 (**term len 23**), and there are 20 lines already displayed. Because there are *less* than 5 lines left in the page, the “---MORE---” prompt is displayed before displaying the next line, and the output pauses for user input at the prompt. (If `automore_conditional()` had *not* been invoked, the “---MORE---” prompt would have been displayed 3 lines later, at the page size of 23.)

In the same case (`automore_conditional(5)` with page size 23), if there were only 10 lines of the terminal length already displayed, in this case there are *more* than 5 lines left in the page, so the “---MORE---” prompt would not be displayed and the output would not pause.

This function can be used to better control when automore processing stops the output, for example, to intentionally delimit a particular segment of displayed information, or to more logically break up the output.

17.1.10 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 17-2 lists the parameters that the `%C` and `%CC` descriptors require. Table 17-3 lists the `printf()` modifiers that the `%C` and `%CC` modifiers support.

Table 17-2 Time-String Descriptor Formats

Descriptor Format	Description
<code>char *</code>	Format string. See the <code>format_time()</code> function in the “ Time-of-Day Services ” chapter in the <i>Cisco IOS API Reference</i> for details about this string.
<code>clock_epoch *</code>	Time epoch to convert.

Table 17-3 %C and %CC Descriptor Modifiers

Modifier	Description
<code>nn</code>	Specifies the field width.

Table 17-3 %C and %CC Descriptor Modifiers (continued)

Modifier	Description
-	Right-justifies the field.
#	Formats the time as UTC rather than in the local time zone.

17.1.10.1 Examples: Format Time Strings

This section provides several examples of how to format time strings.

In the following code, the %CC 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 %CC", "%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 %CC, 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
```

17.1.11 Format Timestamps

To format a timestamp, use the T modifier (possible with other modifiers) followed by a single format code. Table 17-4 lists the `printf()` modifiers and Table 17-5 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 17-4 printf() Modifiers for Timestamps

Modifier	Description
-	Right-justifies the field. This modifier is meaningful only if you specify a field width.

Table 17-4 printf() Modifiers for Timestamps (continued)

Modifier	Description
#	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 17-5 print() Format Codes for Timestamps

Format Code	Description
TA	Formats an absolute time, printing the numerical value of a timestamp. The argument is a 64-bit <code>sys_timestamp</code> .
Ta	
TC	Formats a difference between two timestamps, printing the number of centiseconds. The argument is a 32-bit <code>ulong</code> .
Tc	
TD	Formats a difference between two timestamps, printing the number of milliseconds. The argument is a 4-bit <code>sys_deltatime</code> .
Td	
TE	Formats an elapsed time, printing the time elapsed since the timestamp. The argument is a 64-bit <code>sys_timestamp</code> .
Te	
TF	Formats a future time, printing the time remaining until the timestamp. The argument is a 64-bit <code>sys_timestamp</code> .
Tf	
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.
Tg	
TM	Formats a difference between two timestamps, printing the number of milliseconds. The argument is a 32-bit <code>ulong</code> .
Tm	
TN	Formats the current time, printing the numerical value of the current time.
Tn	This format takes no argument.
TS	Formats a difference between two timestamps, printing the number of seconds. The argument is a 32-bit <code>ulong</code> .
Ts	

17.1.11.1 Examples: Format Timestamps

Table 17-6 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 *yh*—Hours
- *xd*—Days
- *xw*—Weeks
- *xy*—Years

Table 17-6 Examples of Formatting Timestamps

Print Format Specification	Print Output Format
%TE	<i>hh:mm:ss xdyzh xwzd xyzw</i>
%TF	
%TG	
%ITE	<i>hh:mm:ss.mmm xdyzh xwzd xyzw</i>
%#TE	If the time to be displayed exceeds a year, it will display as years and weeks (xyzw). For example: 2y8w If the time is less than a year but more than a week, it will display as weeks and days (xwzd). For example: 7w5d If the time is less than a week but more than a day, it will display days and hours (xdyzh). For example: 3d22h or 3d02h If the time is less than a day but more than an hour, it will display hours and minutes (<i>hh:mm</i>). For example: 08:22 If the time is less than an hour, it will display minutes only (with NO seconds) (<i>m</i>). For example: 23
%Te	<i>s</i>
%Tf	
%Tg	
%lTe (Note that the character after % and before T is the letter l, not the number one.)	<i>s.ms</i>
%#Te	The %#Te format prints the output in hex, eg: if the elapsed time is 1 second, the %#Te format will display 0x3E8 (which is 1000 ms).
The listed formats should all look alike for all the types (A, C, D, E, F, G, M, N, and S); all that matters is whether the format letter is upper- or lower-case, and the presence or absence of the leading 'l' and '#'. Not all these combinations make sense, but the code supports them.	

Table 17-7 shows examples of the output for various time values.

Table 17-7 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

Table 17-7 Examples of Timestamp Output (continued)

Value	Time	Output
6000000000	9 weeks plus	9w6d
600000000000	1 year plus	1y47w

17.1.12 Format AppleTalk Addresses

The `printf()` function provides two format codes specific for formatting AppleTalk addresses, `%a` and `%A`.

17.1.12.1 %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 17-8.

Table 17-8 printf() %a Conversion Flags

Conversion Flag	Description
-	Ignored.
0	Ignored.
+	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.
1	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 1 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 17-9 shows examples of using the conversion flags to format the value 0xab0245.

Table 17-9 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

17.1.12.2 %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 17-10.

Table 17-10 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 1 codes.
nn	Minimum field width. Spaces are used to fill the field to this width.
*	Reads the field width from the next parameter.
1	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 1 codes. This flag is ignored if you specify #.

If you do not specify the optional # or 1 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 form 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 1 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 1 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 17-11 shows examples of using the conversion flags to format the values 0x12345678 and 0x123456.

Table 17-11 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.

17.1.13 Format Banyan VINES Addresses

The `printf()` function provides two format codes specific for formatting Banyan VINES addresses, `%z` (lowercase z) and `%Z` (uppercase Z).

17.1.13.1 %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 17-12.

Table 17-12 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.

Table 17-12 printf() %z Conversion Flags (continued)

Conversion Flag	Description
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 or 0000004144:00048 if **vines decimal** is enabled.

17.1.13.2 %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 17-13.

Table 17-13 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.

17.1.14 Format IPv6 Addresses

The `printf()` function provides one format code, `%P` (uppercase P) for formatting IPv6 addresses.

17.1.14.1 %P Format Code

The `%P` (uppercase P) code formats one parameter, a pointer to an `in6_addr_t`, as an IPv6 address. No conversion flags are supported. Examples are shown in Table 17-13.

Table 17-14 Examples of IPv6 Addresses

Input Address	Displayed as
1080:0:0:0:8:800:200C:417A	1080::8:800:200C:417A
FF01:0:0:0:0:0:101	FF01::101
0:0:0:0:0:0:1	::1
0:0:0:0:0:0:0:0	::

Exception Handling

18.1 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.

Note Cisco IOS exception handling questions can be directed to the interest-os@cisco.com alias.

18.2 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 18-1 presents the full list of exceptions.

Table 18-1 Exception Signals

Signal	Description
SIGABRT	Used by abort
SIGALRM	Alarm clock
SIGBUS	Bus error
SIGCLD	Death of a child
SIGCHLD	
SIGSEGV	Segmentation violation
SIGEMT	EMT instruction
SIGEXIT	Sent just prior to process destruction

Table 18-1 Exception Signals (continued)

Signal	Description
SIGFPE	Floating-point exception
SIGHUP	Hangup
SIGILL	Illegal instruction
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

18.3 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.

18.3.1 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.

18.3.1.1 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);
```

18.3.2 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.

18.3.2.1 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;
}
```

18.4 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.

18.4.1 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);
```

Writing Cisco IOS Error Messages

Added text to section 19.1 “Error Message Guidelines”. (April 2011)

The Cisco IOS error message facility lets the development engineer use the `msgdef()` function to define an error/system message and use the `errormsg()` function to report errors and changes in system status or condition. When an error occurs during operation, the operating system sends these error messages to the system console and, optionally, to a local buffer, a TTY/VTY, or a logging server on another system. Error messages help Cisco customers and customer engineers in the TAC identify the type and severity of a problem.

Note This chapter describes error message services in Cisco IOS Release 11.2 and later. For more information on error message services prior to Cisco IOS Release 11.2, see CSCdi67083 and ENG-77462, “Writing Cisco IOS Error Messages”.

Note Cisco IOS error message development questions can be directed to the `ios-errmsg-review @cisco.com` alias.

It is important to use the Cisco IOS error message service to report errors for several reasons:

- Clear and informative error messages make it easier for customers and Customer Advocacy (CA) engineers to understand system conditions and to correct system errors.
- Clear and informative error messages can reduce calls to TAC, and indirectly to Engineering, thus saving Cisco money.
- Error messages are sent to the system console. The user can also configure the system to log the errors. Logged messages can be stored in an IOS-based log and on a remote logging (`syslog`) server. Logged error messages can even generate SNMP traps.
- Consistent error messages make it easier for customers to identify error conditions by parsing system log files for specific error messages.
- Technical writers add error messages to the *Cisco IOS Software System Messages* mainline reference books and to various platform-specific reference books. Cisco customers use these books to identify errors and solutions. TAC representatives also use these books to help customers to troubleshoot.

The Error Message Decoder (EMD)

(<http://www.cisco.com/cgi-bin/Support/Errordecoder/home.pl>) and Output Interpreter (OI)

(<https://www.cisco.com/cgi-bin/Support/OutputInterpreter/home.pl>) tools extract the messages from the books and display them in the respective tool on TAC Web. Other tools, such as Google-based search, are also used to extract the error messages.

Note Cisco IOS error message development questions can be directed to the ios-errmsg-review@cisco.com alias.

This chapter includes the following sections:

- Error Message Guidelines
- Submitting Error Messages for Review
- Defining an Error Message
- Testing the Error Message
- Generating Error Messages
- Coding Error Messages Example

19.1 Error Message Guidelines

Error messages that have a clear explanation of the error and useful suggestions of actions for the customer to take can reduce the number of technical support calls to the TAC. Detailed, well-written error messages can also reduce the amount of time that the CA engineer must spend to resolve the problem. Useful error messages can cut down the number of contacts that you as the development engineer receive from the TAC, saving DEs' time in the long run.

Follow these guidelines when reporting errors:

- 1 Use the error message services to report errors.

Do not use `printf()` or `buginf()` to report error messages. When using an error message, the user is provided information about how to resolve the issue which is available to the user via the documentation relative to their release and platform as well as via the EMD and OI tools. When an error message isn't used the user cannot easily find further information about the issue.

Note The `printf()` function should still be used when the error message is in direct response to a command (e.g. the user typed a command and one of the arguments is invalid). Also, the `buginf()` function should still be used when a debugging flag controls the `buginf()` output. The debugging flag is enabled or disabled by entering the **logging buginf on** or **logging buginf off** command, respectively.

- 2 Make the information in the error message as clear and accurate as possible.

Clarify the cause of the error and, if appropriate, the severity of the error in the "Explanation" section. Document the actions that should be taken by the customer and the TAC engineer in the "Recommended Action" section. Do not simply tell the customer to contact Cisco. Do not use non-human-readable text such as ASIC names, hex values, or code routines (unless this information is valuable to the customer reading the errmsg). Do not include any information that would reveal any proprietary parts of the Cisco IOS code base, such as field and routine names.

- (a) Write messages that describe what's going on in some detail. What data structures are you acting on? Provide useful state information and context information. Describe what message or event caused the action? Provide some detail of exactly what went wrong?
- (b) Don't use identical messages for different conditions. If the message is in an inline and therefore shared, pass into the inline some context information (other than source code details) for additional clarity.
- (c) If necessary, you can always print the PC value or a stack trace. This can be translated to specific code locations by anyone who has access to the source repository.

3 Define an error message facility for a specific component or service.

A feature can have several different error message facilities. And several `msg_xxx.c` files can exist in a directory.

4 Define one or more error messages for a facility. Do not overuse an error message.

Each type of error within a facility should have its own error message. When one error message is used to handle various errors, such as when you define a message whose text is simply "%s", the various causes of the error cannot be properly documented. For example, messages like `msgdef(ERROR, FOO, ..., "%s")` cannot be properly documented.

Avoid using a generic error message to describe a variety of error conditions. If multiple types of errors could occur in your code, create multiple error messages.

5 Verify that the file has its interest attribute set in ClearCase as shown using the `cc_interest` ClearCase command. This can be done using the file ownership tool at http://wwwin-eng.cisco.com/Eng/IOS/Tools/File_Ownership/ or using the ClearCase `cc_interest` command. A component is associated with the interest attribute and is required so that technical writers and Customer Advocacy know who to contact for clarification of a message.

6 Verify that the `msgdef_ddts_component()` is set for the error message.

7 Try to make error message facilities consistent. For example, use "C6K" as the start of all Catalyst 6000-related messages (C6K_LC, C6K_OSM_ATM, etc.).

8 Keep the error message consistent from Cisco IOS release to release. Customers that are using several Cisco IOS releases can be confused if they see different error messages describing the same problem.

9 As a rule, do not report `malloc()` failures with `errmsg().malloc()` creates its own rate-limited error message to report failures.

If, for example, a user had just entered a command to enable a routing protocol and then received a `SYS-2-MALLOCFAIL` message, the user would see the connection between the action and the failure report without receiving an additional `errmsg()` and would most likely be aware that the routing protocol in question would not function properly due to the failure.

It may be appropriate to display an `errmsg()` that is related to the `malloc()` failure. For instance, when insertion of a module results in having a `malloc()` be done and the `malloc()` fails, it may be appropriate to display an `errmsg()` if it has an explanation adequately linking it to the `malloc()`. For example, "The OIR may have failed due to lack of memory <or some other reason>. If you saw a %SYS-2-MALLOCFAIL message, this indicates the error was due to lack of memory".

10 Do not create an `errmsg()` that will produce a lot of output in a short time that would overwhelm the console or logging buffers. Use `msgdef_limit()` to rate-limit messages that might produce a lot of output in a short time.

19.1.1 What to Avoid in Error Messages

- 1 Avoid the use of the word “fatal”, for example:

```
%ERR-1-GT64010: Fatal error, Memory parity error (external)
    cause=0x0100E283, mask=0x0CD01F00, real_cause=0x00000200
    bus_err_high=0x00000000, bus_err_low=0x00000000,
    addr_decode_err=0x14000470
```

Also, the `cause` and `real_cause` values are in hex only and are not human-readable.

GT64010 exposes internal, ASIC-level detail.

It is not clear what “external” refers to, making it difficult to determine which specific field-replaceable unit (FRU) needs to be replaced (i.e. only a piece of memory or the full adapter or route processor?). It is not clear from the message what the user needs to do next to recover and/or protect their network from another occurrence of the problem.

- 2 In the following message, “reloading” suggests that the system is recovering on its own and no further user action is required. Clarification in the `msgdef` file itself could provide further direction to the user.

```
%ERR-1-SERR: PCI bus system/parity error
%ERR-1-FATAL: Fatal error interrupt, reloading
    err_stat=0xA400, err_enable=0xE303
```

Although “`err_stat`” and “`err_enable`” appear relevant to determining what the user should do next to protect their network long-term from a similar event, they are in hex only and are not human-readable.

19.2 Submitting Error Messages for Review

Before committing new error/system messages or error/system message modifications, send the changes to the `ios-errmsg-review` email alias for an editorial review.

List the reviewer from `ios-errmsg-review` in the “Reviewer” field of the DDTs.

Please allow seven calendar days (five business days) for a review. Usually, the review is much quicker (within two days, or before the end of the next day). If you require a faster review, please say so in your message to the alias.

19.3 Defining an Error Message

19.3.1 Error Message Files

Error messages are defined in a `msg_xxx.c` file. For example, the `msg_c10k_atm.c` file defines error messages for the ATM line card for the Cisco 10000 platform.

Each `msg_xxx.c` file has several important components that are shown in Table 19-1:

Table 19-1 msg_xxx.c Components

Component	Description
facdef()	Macro that defines the facility name. A facility is a group of related error messages.
msgdef_section()	Identifies the section of the file that defines error messages for the facility.
msgdef()	Macro that defines the specific error message by providing a name, severity, and error message string. The error message name is also known as the mnemonic. The facility/mnemonic pair uniquely identifies an error message. The facility, severity, mnemonic, and string constitute the error message that is displayed when the error occurs. The error message also appears in the <i>Cisco IOS Software System Messages</i> manual.
msgdef_limit()	Macro that defines a rate-limited error message.
msgdef_explanation()	Macro that defines the error message explanation. This text appears in the “Explanation” section of the error message’s entry in the <i>Cisco IOS Software System Messages</i> manual. It is important to avoid using “non-human-readable” information such as ASIC names, HEX values, or other formats that don’t have meaning for the customer (unless this information is valuable to the customers reading the message).
msgdef_required_info()	Macro that defines the information that the customer must gather and provide to the CE on the initial support call. Having all the necessary information up front cuts down on the number of phone calls and the number of frustrated customers.
msgdef_recommended_action()	Macro that defines the action that the user should take. This information appears in the “Recommended Action” section of the error message’s entry in the <i>Cisco IOS Software System Messages</i> manual. This information should include suggested show commands, and troubleshooting steps to help the customer resolve the issue. If for any reason the customer can’t solve the issue, the Recommended Action should suggest what information to collect before the customer opens a case through TAC.
msgdef_ddts_component()	Macro that identifies DDTs component producing the error. The msgdef_ddts_component should typically be the same component that is in the interest list.
msgdef_tac_details()	Macro that defines additional information about the error condition that the CA engineers can use to help diagnose the problem. This information does not appear on the system console or in customer reference books, but is available to TAC representatives. You can also provide advice on how to clarify whether this is a hardware or a software issue. Because customers never see the contents of this macro, you can include sensitive information, such as hidden engineering commands, for TAC use.

Defining an Error Message

19.3.1.1 msg_*.c files That Have Been Replaced by msg_*.rc Files

The following changes were committed into the tuono and aloo branches. Tuono already collapsed into haw_t_pi2_itd1. The .rc files are in this PI and onward now on the T trains. For the S train, it has not been decided as of 03-09-05 whether to collapse aloo into either const2 or flo_isp.

With the SingleSource project, some `msg_*.c` files have been replaced by `msg_*.rc` files. Currently in Cisco IOS, `msg_[foo].c` files play a dual role both as a header file and as a source file. While addressing the monolith dependencies for the `eigrp` component, a few `msg_*.c` files needed to be relocated to the pseudo components. The SingleSource component directory structure assumes that all the source files are in a component's `src` directory and public header files are in the `include` directory. Thus it's not clear if the `msg_*.c` files should be in the `comp/include` or `comp/src` directory.

Therefore, a new convention of `msg_system.rc` files has been set up that uses the **rc-tool** provided by the component build system. An `.rc` file is both a header file and a `.c` file. It can be included just like a header file and compiled just like a `.c` file. When a `.rc` file is compiled, it generates the definitions and when it's being included, it generates only the externs.

Note The rc-tool is just a build system tool that has been integrated into the component build system. If a file is a `.rc` file and is being compiled, then the rc-tool helps generate the message definitions. If the `.rc` file is only being *included* by a file, then the rc-tool generates only the externs for the message definitions. In other words, users don't have to execute any command. The `msg_*.rc` file format gives us the capability of maintaining the message definitions in one file as opposed to the existing system where we have to maintain two `msg_*.c` and `msg_*.h` files. It also eliminates other Cisco IOS files from having to include a `.c` file for the message definitions.

The following changes took place:

- `/vob/ios/sys/os/msg_sched.c` and `os/msg_sched.h` were replaced by
`/vob/ios.comp/kernel/sched/include/msg_sched.rc`
- `/vob/ios/sys/xns/msg_ipx.c` was replaced by
`/vob/ios.comp/xns/include/msg_ipx.rc`
- `/vob/ios/sys/os/msg_system.c` and `/vob/ios.comp/kernel/include/msg_sys.h` were replaced by `/vob/ios.comp/kernel/include/msg_system.rc`
- `/vob/ios/sys/iprouting/msg_iproute.c` were replaced by
`/vob/ios.comp/iprouting/include/msg_iproute.rc`

19.3.2 Defining Error Messages in the Error Message File

The error message service provides a set of macros and functions to define one error message facility and one or more error messages in a `msg_xxx.c` file.

Note Error messages and debugging messages may be dropped before being displayed with no indication in the trace that the messages have been lost other than possibly a gap in the timestamps. This will only occur if the number of messages held in the display queue reaches a pre-specified maximum. That maximum value is initially set by the `platform_get_value(PLATFORM_VALUE_LOG_MAX_MESSAGES)` platform-specific routine. It may also be overridden by the hidden **logging queue-limit** configuration command.

19.3.2.1 Setting up #include's and #define's

Include only the following in the `msg_xxx.c` file:

```
#include "master.h"
#define DEFINE_MESSAGES
#include "logger.h"
```

Note The `#define DEFINE_MESSAGES` statement must precede the `#include "logger.h"` statement. If `DEFINE_MESSAGES` is not defined, then `facdef()`, `msgdef()`, and `msgdef_limit()` do nothing.

19.3.2.2 Defining an Error Message Facility

Use the `facdef()` macro to define an error message facility. An error message facility defines a group of related error messages. The `facdef()` macro takes the facility name as an argument.

For example, sample facilities can be found in the table, “Table 1 System Error Message Facility Codes” in the following document:

<http://www.cisco.com/univercd/cc/td/doc/product/software/ios122s/122sdebu/sem122sa.htm>

Follow these guidelines when defining a facility:

- Use at least two initial uppercase letters to define facilities. You can also use underscores and digits in the facility name, but no hyphens. Do not use lower-case letters in the facility name to avoid case sensitivity issues.
- Make the facility name unique among all facility names in the Cisco IOS or non-IOS platform code base.
- Do not use Cisco code names (such as internal names for ASICs or modules), specialized acronyms, or technical jargon that the customer will not recognize. Never use slang or humor. For example, in the following: `GEN7_ACL-3-ADD_FEAT_FAILURE`, `GEN7` exposes the internal code name for the 12000’s Engine 4+ line card.

Instead of using an ASIC name, try to use a brief description of the ASIC’s functionality as the user would perceive it. For example, on Catalyst 6500, instead of “`MEDUSA`”, “`C6K_SFM_INTERFACE ASIC`” could be used, because customers are familiar with the acronym “`SFM`” meaning “Switch Fabric Module”.

- Define only one facility in the `msg_xxx.c` file. You can define one or more error messages within a facility. The following example defines the IPC facility for the Gigabit Route Processor on the Cisco 12000 platform:

```
facdef(IPCGRP)
```

19.3.2.3 Defining Error Messages

Use the `msgdef_section()` macro to define the message definition section of the `msg_xxx.c` file. Pass an explanation string in to this macro. The `msgdef_section()` forms a title for the facility in the error message documentation. The `msgdef_section()` definition follows the `facdef()` declaration.

Use the `msgdef()` macro to define an error message or `msgdef_limit()` macro to define a rate-limited error message.

Note Error messages and debugging messages may be dropped without being displayed, with no record of loss in the trace other than possibly a gap in the timestamps. This will occur only if the number of messages held in the display queue reaches a specified maximum. That maximum value is initially set by the `platform_get_value(PLATFORM_VALUE_LOG_MAX_MESSAGES)` platform-specific routine. It may also be overridden by the hidden **logging queue-limit** configuration command.

If the message is for a particular platform, the description should include the platform's proper name as shown on www.cisco.com. Engineering and marketing should collaborate early in the development process to ensure the released name is incorporated into the messages.

The `msgdef()` macro takes the 5 parameters shown in Table 19-2:

Table 19-2 msgdef() Parameters

Parameter	Description
Name	Is the name of the message. The name should be easy to remember, unique within the codebase, and have at least two uppercase characters. The name is also known as the error message mnemonic. Characters allowed in the name are A-Z (upper case alphabetical characters) and 0-9. Do not use lower-case letters in the facility name to avoid case-sensitivity issues.
Facility	Is the facility defined by the <code>facdef()</code> macro described above. The facility name should have at least two uppercase letters.
Severity	Is the severity of the error. Table 19-3 describes error message severities.
Flags	<p>Is 0 or one or two error message flags.</p> <p>MSG_TRACEBACK—A PC trace is included with the error message. This trace has saved PC values for up to eight routines saved on the stack.</p> <p>MSG_PROCESS—Information about the currently running process is included with the error message. This information includes the process name, processor priority level, and process ID. This flag is safe to use on messages in interrupt service routines.</p> <p>MSG_CLEAR—Indicates that a previously-reported condition has been cleared. A “clear” condition can also be reported by using an <code>errmsg</code> with the same severity as the original <code>errmsg</code> that reported the error.</p> <p>MSG_NOSCAN—Prevents the Embedded Event Manager <code>syslog</code> Event Detector from performing a regular expression pattern match scan.</p> <p>Pass in 0 if no flags are set. Use a pipe character () between flags if more than one flag is used.</p>
Format	<p>Is a <code>printf()</code> style format string for the message. This string is printed out as part of the error message. This string should also not print out non-human-readable information, such as ASIC names, hex values or code routines (unless this information is valuable to the customer reading the <code>errmsg</code>). This information causes confusion for the customer.</p> <p>Although the <code>printf</code> string style format allows <code>\n</code> to be used in the message context, it is advised that the message designer should avoid using it because any content after <code>\n</code> can be ignored by some scripting applications deployed at Cisco. Currently, known scripting applications, such as Network Management scripts and applications, will ignore any content after <code>\n</code> and therefore can be broken.</p>

The combination of message facility and message name creates a unique global symbol for the message.

The following example defines the error message definition section for IPC routing:

```
msgdef_section("Gigabit Route Processor (GRP) Interprocess Communication
(IPC) error messages");
```

The following `facdef()` and `msgdef()` example defines the IPC/NOMEM error message and includes an error traceback:

```
facdef(IPCGRP);
...
msgdef_limit(SYSCALL, IPCGRP, LOG_ERR, MSG_TRACEBACK, "System call for
command %u (slot%u/%u) : %s (Cause: %s);
```

Eight error message severity values exist, as shown in Table 19-3:

Table 19-3 Error Message Severity Values

Severity	Value	Description
LOG_EMERG	0	<p>Indicates that the system is unusable, or has recovered from an unusable condition and is now usable. Reasonably common circumstances cause the entire system to fail, or a major subsystem to stop working, or other devices on the network to be disrupted, and there's no workaround.</p> <p>The following are examples:</p> <pre>System shutting down due to total fan tray failure System shutting down due to missing fan tray Environmental Monitor initiated shutdown</pre> <p>Corresponding messages reporting that the system has recovered can have the same severity level; for example:</p> <pre>System shutdown canceled: a fan tray has resumed normal operation</pre>
LOG_ALERT	1	<p>Indicates that immediate action is needed.</p> <p>The following are examples:</p> <pre>Core temperature CRITICAL limit exceeded Failed to configure [dec] interfaces in bay [dec], shutting down bay</pre> <p>Corresponding messages reporting that the system has recovered can have the same severity level; for example:</p> <pre>Core temperature CRITICAL limit returned to normal</pre>
LOG_CRIT	2	<p>Indicates that a critical condition exists or no longer exists. Important functions are unusable, and there's no workaround, but the router's other functions and the rest of the network are operating normally.</p> <p>An example would be IP helper addresses being ignored.</p>

Defining an Error Message

Table 19-3 Error Message Severity Values (continued)

Severity	Value	Description
LOG_ERR	3	<p>Indicates that an error condition exists or no longer exists.</p> <p>Something has failed under unusual circumstances, or minor features don't work at all, or something has failed but there's a low-impact workaround.</p> <p>The following are examples:</p> <pre>HTTP out of memory</pre> <pre>LAPB timer task cannot be created</pre> <pre>Bad IP address or host name [chars]</pre> <p>This is the highest level for documentation bugs.</p>
LOG_WARNING	4	<p>Indicates that a warning condition exists or no longer exists.</p> <p>Something has failed under very unusual circumstances and recovered as if by itself. Users don't need to install any workarounds, and performance impact is tolerable.</p> <p>An example would be that the first segment for each TCP connection is always retransmitted.</p>
LOG_NOTICE	5	<p>Indicates that an event has occurred that has no real detrimental effect on system functionality.</p> <p>The following are examples:</p> <pre>Subinterfaces are not supported on FastEthernet</pre> <pre>Nonvolatile storage configured from [chars].</pre>
LOG_INFO	6	<p>Provides system state information.</p> <p>The following is an example:</p> <pre>PKCS #12 Successfully Exported card in slot [dec] enabled</pre>
LOG_DEBUG	7	<p>Provides debugging information.</p> <p>The following is an example:</p> <pre>Reading [hex] from carrier</pre> <pre>ISDN outgoing called number: [chars]</pre>

Including the corresponding failure recovery messages at the same logging level is valid because customers may have syslog configured to view messages at a particular severity. Failure recovery messages may not be seen unless they are at the same logging level as failure messages.

The severity level assigned to an error message should agree with the recommended user action. Lower-severity errors should include actions such as correcting a hardware problem, an environmental condition, or a misconfiguration. Higher-severity levels may include information-only and debug statements. If a message is in the gray area between those that require action and those that do not, include a threshold at which point action is required.

If the message truly indicates an error, state in the message whether the error is a corrected error or an error needing user action. The following example clarifies for the user that the error is “corrected” and thus has set the severity level at 6.

```
EE48-6-ECC: Corrected error in Rx Alpha TLU SDRAM at address x.
```

In contrast, the following messages are reporting errors as being corrected, but the customer cannot see the outcome reported from the message itself. Avoid this kind of vagueness.

```
%Tiger-3-SBE: Single bit error detected and corrected at <address>
%TX192-3-SDRAM_SBE: Error=0x2 - DIMM1 Syndrome=0x7600 Addr=0xBEA09 Data bit80
%LC-3-ECC: Salsa ECC: Addresses: Salsa returned = 429BFDE8 correcting on =
429BFDE8
```

If an error has been corrected, state the fact explicitly in the message; don't rely on a severity of 6 to convey that information to the customer.

For information on GSR soft error reporting see EDCS document 266795, *GSR Soft Error Reporting Requirements*.

Table 19-4 illustrates error messages that have good format values.

Table 19-4 Example of Error Messages with Good Format Values

Error Message	Customer Advocacy's Evaluation
%FIB-2-FIBDOWN: CEF has been disabled because of a low memory condition. It can be re-enabled by configuring ip cef [distributed]	Provides a next step and suggested show/clear commands to use for recovery.
%MDS-2-RP: MDPS is disabled on some line card(s). Use "show ip mds stats linecard" to view status and "clear ip mds linecard" to reset.	
%ATM-3-FAILCREATEVC: ATM failed to create VC(VCD=3503, VPI=0, VCI=1076) on Interface ATM1/0, (Cause of the failure: Failed to have the driver accept the VC)	Prints a human-readable cause.

Table 19-5 illustrates error messages that have bad format values.

Table 19-5 Example of Error Messages with Bad Format Values

Error Message	Customer Advocacy's Evaluation
SLOT 1:Aug 18 21:03:48.995 jst: %LC_SSRP-3-SOP: RX:RsrdBitsUsed. Source=0x1(Plim), halt_minor0=0x0	Expose the internal ASIC name (SOP and BMA). “RsrdBitsUsed” is unclear.
%LC-3-BMA48ERRS: [chars] BMA48 [chars] error [hex]	Uses a hex value and no human-readable value in “halt_minor” field.
%LC-3-BMACMDRPLY: Problem in BMA reply to command type 128 ToFab BMA sequence no=64	Identifies the command number. Users are not clear whether the number is relevant or simply a sequence number.
%IPCGRP-3-CMDOP: IPC command 40 (slot1/0): linecard is disabled	Unclear what to do to recover if you see this message.
%LC-3-BMAERRS: FrFab BMA BMA error status error 8	Error status field uses a hex value and no human-readable value.

Defining an Error Message

Table 19-5 Example of Error Messages with Bad Format Values (continued)

Error Message	Customer Advocacy's Evaluation
%LC-3-BMA48ERR: FrFab BMA error: stat_reg 300B2 adr 0 data 0 qm 1872F98 plim 0 13 0ms 0 sdram 0	Full register dump without any human-readable clarification.

The `msgdef_limit()` macro also defines an error message. This macro takes the five `msgdef()` parameters and an additional rate-limit parameter that defines the minimum interval in milliseconds between repetitions of the same error message.

Four rate-limit time macros are defined in Table 19-6:

Table 19-6 Rate-limit Macro Time Values

Macro name	Value
MSGDEF_LIMIT_GLACIAL	1 minute
MSGDEF_LIMIT_SLOW	30 seconds
MSGDEF_LIMIT_MEDIUM	5 seconds
MSGDEF_LIMIT_FAST	1 second

The following example defines an `IPCGRP SYSCALL` message that the Cisco IOS software will not send more than once every five seconds.

```
msgdef_limit(SYSCALL, IPCGRP, LOG_ERR, MSG_TRACEBACK,
    "System call for command %u (slot%u/%u) : %s (Cause: %s)",
    MSGDEF_LIMIT_MEDIUM);
```

If you are creating a bulleted list, use the `\n` character along with a dash. Here is an example using the Gatekeeper `errmsg` facility:

```
msgdef(GKTMP_SERVER_OFFLINE, GK, LOG_ERR, 0,
    "GKTMP Server: %s(%s) is disconnected with Gatekeeper: %s");
msgdef_explanation(
    "The connection between a GKTMP server and a Gatekeeper "
    "has been closed. The reason for the closed connection "
    "could be one of the following:\n- The Gatekeeper has "
    "initiated a connection closure because a socket read or "
    "write operation failed;\n- A TCP connection may "
    "be broken;\n- The Gatekeeper has received an "
    "unrecognized GKTMP message; \n- The GKTMP server has "
    "been unconfigured on the Gatekeeper by user-entered "
    "CLI commands;\n- The trigger has been administratively "
    "shut down on the Gatekeeper by user-entered CLI "
    "commands; \n- The GKTMP server has closed the "
    "connection with the Gatekeeper.");
```

19.3.2.4 Creating an Error Message Explanation

Use the `msgdef_explanation()` macro to explain the error. Pass an explanation string in to this macro.

The explanation will be extracted and will appear verbatim in the “Explanation” section of this error message in the *Cisco IOS Software System Messages* document. Make the explanation clear and concise to avoid confusing the customer. This information is not displayed by the Cisco IOS system.

The following example creates an error message explanation for the `IPCGRP SYSCALL` error.

```
msgdef_explanation(
    "An IPC kernel system call error has occurred. This "
    "condition might have been caused by an IPC message that "
    "was sent from the RP to the LC and that blocked other "
    "IPC messages. The RP sets a timer waiting for a response. "
    "If it does not receive a response in the given timeout "
    "period, the RP will generate the above traceback message.");
```

The text should be quoted as in the above example, with each line beginning with a quote and ending with a quote (having a string span multiple lines is not valid in ANSI C). Indent the text to one tab stop.

An error message explanation should contain the following elements:

- The *component* that is involved.

If the component is a commonly-known networking term that customers are familiar with, APPN or BGP for example, the component might be the same as the first section of the mnemonic and require little additional explanation. Examples of commonly-known networking terms are APPN (APPN subsystem) and BGP (BGP subsystem).

If the component is not a commonly-known networking term that customers would be familiar with, for example, GPRSFLTMG or HAWKEYE, there should be additional clarification about the component. For example, if the message is BLIVET-1-CRASH, explain what BLIVET is by stating “The BLIVET super-interface card” in the explanation. If the external marketing name is known, use the marketing name for the component.

- The *category* of the message.

The category should be more specific than “hardware”, “software”, “error”, and the like. Examples of more specific categories are in Table 19-7:

Table 19-7 System Message Categories

Category	Category Types
Hardware	Power (too little, too much) Cable missing Hardware missing (card missing) Hardware inadequate (too little memory) Hardware failing (card failing) Hardware incorrect (wrong type of card) Too many components (too many cards in the system) Status information (powering off, powering on, cards coming online)
Software	Configuration error by user Coding error
Environmental	Temperature too cold Temperature too hot

Defining an Error Message

Table 19-7 System Message Categories (continued)

Category	Category Types
External	Traffic-related (too much, too little) Duplicate IP addresses Neighbor disappearing

- The *significance* of the message.

How does the message condition affect system operation? Will the system attempt to fix the problem? Should the user be overly concerned by the message (if this is indicative of a hacker attack, for example)?

- The *frequency* of the message.

Is the frequency of the message significant? If it happens once during boot-up or some other time, can it be safely ignored if it does not recur?

Put these elements together in an explanation, as shown here:

The *component* has been *category*. *significance* and *frequency*.

For example:

The *super-interface card* has been *detected during hardware initialization*. This message is informational only and should occur once during system initialization.

The following explanations are not specific enough:

- Repair or replace the controller. (this is not an explanation, it is an action; and the TAC cannot repair hardware)
- None
- Unknown
- An error has been reported by the firmware.
- An internal software error has occurred.
- A hardware or software error has occurred.
- No action is required. (this information belongs in the recommended action)
- Information message only. (this information belongs in the recommended action)

19.3.2.5 Specifying a Recommended Action

Use the `msgdef_recommended_action()` macro to suggest an action for the user to perform when the error occurs. Pass a string to this macro. Include any commands or other effective actions a user can perform to diagnose the problem and to fix it. If a command is included, use the `<CmdBold>` and `<NoCmdBold>` tags around the command so it will appear in boldface when it is extracted from Cisco IOS code and is documented in Cisco IOS documentation.

Instruct the customer what to do when the message occurs, for example:

- Instruct the user to issue **show**, **debug** or **clear** commands.

show commands: which **show** commands are required to resolve or observe the condition?

If the **show tech-support** command is required, what part of the **show tech-support** output is of interest? (**show tech-support** contains the following commands: **show version**, **show running-config**, **show controllers**, **show stacks**, **show interfaces**, **show context**, **show diag**, **show region**, **show processes memory**, **show processes cpu**, and **show buffers**.)

debug commands: which **debug** commands are required to resolve or observe the condition?

clear commands: which **clear** commands are required to resolve or observe the condition?

- Refer the user to documentation on www.cisco.com.

If the problem can be due to a software configuration, for example, what area of the documentation should the customer refer to? This information does not have to be a URL-specific. For example, if the message could be caused by an OSPF misconfiguration, you could write, “Refer to the OSPF configuration section of the Cisco IOS Documentation”.

If the message could be caused by a hardware misconfiguration, refer the customer to appropriate documentation; for example, a hardware configuration guide for the specific hardware or the software upgrade planner to determine memory requirements.

- Refer the user to bugs that resolve the condition.

If the problem can be due to a bug, the customer should be referred to the Bug Navigator at <http://www.cisco.com/cgi-bin/Support/Bugtool/home.pl> to search for bugs (“caveats” in the IOS world).

- Refer the user to check for environmental issues.

If the message can be caused by factors external to the hardware, the customer should be advised what to look for (room ventilation, cooling, etc.)

- Refer the user to any other miscellaneous actions.

Other actions could include reseating a card, reloading the operating system, or rebooting the system.

- How to deal with messages that are informational only and don’t require any recommended actions.

If the error message is simply an information message, state this and indicate that no action is required. The standard wording for informational messages is as follows:

“This is an informational message only. No action is required.”

or

“The system should recover. No action is required.”

19.3.2.5.1 Calling TAC - The Last Resort

Calling TAC should be the very last resort recommended to the customer. If the customer is instructed to call the TAC, the customer should have been asked to perform some action prior to calling the TAC. The following are examples of poor messages because they do not list customer actions aside from calling the TAC:

- Copy the error message exactly as it appears on the console or in the system log, contact your Cisco technical support representative, and provide the representative with the gathered information. (This is a poor message because it does not state the relevant **show** commands.)

Defining an Error Message

- Copy the error message exactly as it appears on the console or in the system log. Issue the **show tech-support** command to gather data that may help identify the nature of the error. If you cannot determine the nature of the error from the error message text or from the **show tech-support** command output, contact your Cisco technical support representative, and provide the representative with the gathered information. (This is a poor message because it does not tell the customer or TAC engineer what section of **show tech-support** to examine.)
- If this message recurs, copy the error message exactly as it appears on the console or in the system log, contact your Cisco technical support representative, and provide the representative with the gathered information. (This is a poor message because it does not state the relevant **show** commands.)
- Check the version of server software; upgrade if necessary. (This is a poor message because it does not refer the customer to useful web tools, such as the Software Advisor or Bug Navigator to make an informed decision on a software upgrade.)
- A system software upgrade might be required for this module. Contact your technical service representative. (This is a poor message because it does not refer the customer to useful web tools, such as the Software Advisor or Bug Navigator to make an informed decision on a software upgrade.)

The elements necessary in an action would depend on the type of the problem:

- Issue **show** commands, **debug** commands, or **clear** commands to determine the nature of the problem.
- Refer to Cisco IOS or other documentation to correct configuration problems.
- Refer to Bug Navigator to search for bugs.
- Correct the environmental condition, if present.
- Perform miscellaneous actions.
- If the actions listed have not resolved the message, gather the output from the **show** commands, **debug** commands, or **clear** commands prior to contacting the TAC.

A message action would probably not contain all of these elements, but could. For example:

Issue the **show blivet** command to ensure that the blivet card has been detected. If the blivet card is present, issue the **clear blivet**, then **debug blivet** commands. If the message still occurs, refer to the Software Advisor to ensure that the version of code you are running supports the blivet card. If the blivet card is supported, refer to Bug Navigator to search for caveats relating to the blivet card. Check ensure that the chassis fan is running to cool the blivet card. If the chassis fan is running, reseat the blivet card. If the actions listed have not resolved the message, gather the output of the **show blivet**, **clear blivet**, **debug blivet**, **show version** and **show diag** commands. Contact your Cisco technical support representative and contact the representative with the gathered information.

The following is a `msgdef_recommended_action()` macro example:

```
msgdef_recommended_action(
    "If /"no memory/" appears in the error message, add memory to "
    the card specified in the error message and reenable "
    "distributed CEF. If this message recurs after a memory "
    "upgrade, copy the error message text exactly as it appears "
    "on the console or in the system log, enter the "
    "<CmdBold>show tech-support<NoCmdBold> command, contact "
    "your Cisco technical support representative, and provide "
    "the representative with the gathered information.\n "
    "If /"No window message, LC to RP IPC is "
    "non-operational/" appears in the error message, try to "
    "restart CEF on the specified card by entering the "
    "<CmdBold>clear cef linecard <NoCmdBold> slot-number "
    "command in EXEC mode or entering a <CmdBold>microcode "
    "reload<NoCmdBold> global configuration command for the "
    "Cisco 7500 series. (Entering the "
    "<CmdBold>microcode reload<NoCmdBold> command will "
    "cause a traffic interruption of approximately two "
    "minutes.) Entering these commands should temporarily "
    "restore distributed CEF on the card. If the problem persists, "
    "copy the error message as it appears, log onto "
    "www.cisco.com for specific documentation and tools that "
    "are related to the platform and feature in question. "
    "Documentation can be found "
    "at http://www.cisco.com/univercd/home/home.htm "
    "and various tools and utilities for problem solving can "
    "be found "
    "at http://www.cisco.com/kobayashi/support/tac/tools.shtml. "
    "If this message recurs after a memory upgrade, copy the "
    "error message text exactly as it appears on the console or "
    "in the system log, enter the <CmdBold>show tech-support<NoCmdBold> "
    "command, contact your Cisco technical support representative, "
    "and provide the representative with the gathered information. ");
```

The following is an example of an error message that has been improved. The “before” and “after” versions are shown in Table 19-8 for purposes of comparison.

```
%GRP-3-FABRIC_UNI: [chars] ([int])
```

Table 19-8 Error Message Improvements

Before	After
<p>Explanation A problem has been detected between the master RP and a line card over the fabric.</p> <p>Recommended Action Copy the error message exactly as it appears on the console or in the system log, call your Cisco technical support representative and provide the representative with the gathered information.</p>	<p>Explanation A hardware or software problem has been detected over the fabric between the master RP and the card in the slot specified in the error message. This condition could be created by, a software problem or, less likely, a hardware problem.</p> <p>The following list of possible reasons for this message is not all-inclusive:</p> <ul style="list-style-type: none"> 1 A software failure has occurred in which a process has disabled interrupts long enough for 5 consecutive fabric pings to be missed. This condition indicates a problem with the software on the destination card. 2 The RP could not send a fabric ping to the specified line card. If a fabric ping cannot be sent for a certain number of consecutive times, the destination card will be reset. 3 The RP could not send data other than a fabric ping to the specified line card. If the data that could not be sent is anything other than a fabric ping, the destination card will not be reset, but this condition may affect other functionality of the RP; for example, CEF might be disabled. 4 A hardware problem has occurred on the RP or GRP, in the fabric, or on the destination card. If the error was due to a hardware problem with the GRP, other IPC timeout messages would be logged before the fabric ping failure. Entering the show contr fia command on the GRP would show errors, but entering the show contr fia command on the line card would not. <p>Recommended Action If this message recurs, copy the error message exactly as it appears on the console or in the system log, enter the show controller fia command and, if the line card was reloaded, the show context slot slot-number command, contact your Cisco technical support representative, and provide the representative with the gathered information.</p>

19.3.2.6 Specifying a DDTS Component

Use the `msgdef_ddts_component()` macro to specify the DDTS component in which the error occurs. The following example identifies SNMP as the component:

```
msgdef_ddts_component( "snmp" );
```

19.3.2.7 Identifying Information for TAC Engineer

Use the `msgdef_tac_details()` macro only in very special cases to specify actions that the CA engineer should perform to identify or document the error. Put as much information into the `msgdef_explanation` and `msgdef_recommended_action` as possible and attempt to avoid the use of the `msgdef_tac_details` macro whenever possible.

The `msgdef_tac_details` information might include why this problem occurs or pointers to documented and engineering commands that Customer Engineers (CEs) can use to diagnose the problem. Specify whether the CE should file a DDTS when this problem is reported.

Some TAC people cannot access Cisco IOS source files. This macro can provide information that a customer should not see, and also can provide information for engineering commands.

The following is an example of the `msgdef_tac_details` macro:

```
msgdef(UNSUPPORTED, VPA, LOG_ERR, 0,
       "The %s VPA in %s is not supported by the %s module");
msgdef_explanation(
    "The VPA in the given subslot is not supported in "
    "the given VPA carrier module by the current operating system "
    "version. The VPA type may be supported by another VPA "
    "carrier module and/or by a later operating system release.");
msgdef_recommended_action(
    "Ensure that the given VPA carrier module supports the given "
    "VPA type and that the correct operating system release to "
    "support the VPA type in the VPA carrier module is in use. It "
    "may be necessary to use a later operating system release or a "
    "different VPA carrier module. If the VPA type given in the "
    "error message does not match the information shown on the "
    "front of the VPA in terms of interface types or numbers, copy "
    "the error message exactly as it appears on the console or in "
    "the system log, call your Cisco technical support "
    "representative, and provide the representative with the "
    "gathered information and information about the VPA type found "
    "on the front of the VPA.");
msgdef_tac_details(
    "Ensure that the first [chars], which gives the VPA type, "
    "matches the actual VPA type shown on the front of the VPA, "
    "indicating that the VPA type field read from the VPA's IDPROM "
    "is not incorrect/corrupted. If the VPA type given in the "
    "message is correct, the VPA carrier module type given in the "
    "message is correct, and the operating system version running "
    "on the router is documented as supporting the given VPA type "
    "in the given carrier module, this does not indicate a problem "
    "with either the VPA or carrier module. It indicates that what "
    "is documented as supported is not actually supported by the "
    "software, or there is a software error. Either issue should "
    "be easily reproducible. If the VPA type shown on the front of "
    "the VPA does not match that given in the message, the VPA will "
    "probably need an RMA to have its IDPROM reprogrammed or other "
    "corrective actions taken.");
```

19.4 Testing the Error Message

If you change or add a `msg_` file, run `/auto/ses/bin/static_ios` or `prep-commit` to check the validity of the `msgdef` macros in the file. This will check to ensure the correctness of the `msgdef` macros in the `msg_` file. `static_ios` will also check the validity of the `errormsg()` calls in your `.c` files.

Note You no longer have to run `msgdef_search.perl` since all the functionality of this script is in `static_ios` (tools release 7 in production September 2005).

For additional information on **static_ios**, refer to:

http://wwwin-ses.cisco.com/staticAnalysis/static_branch/

And send questions to **ses-support**.

19.5 Generating Error Messages

Use the **errormsg()**, **errormsg_ext()** (*New in 12.2(105)S*), or **errvarmsg()** function to display an error message.

For example, place calls to **errormsg()** or **errvarmsg()** in code when an error condition could occur:

```
if (!protocol) {
    errormsg(&msgsym(NOREGISTER, PPP), protocol);
    return;
}
```

errormsg() and **errvarmsg()** report the error in the following format:

```
%FACILITY-SEVERITY-NAME: <text of message with printf-style arguments
substituted>
```

When calling these routines, pass in an error message that has been formatted with the **msgsym()** macro, passing it the **FACILITY** and **NAME** of the message you wish to display.

msgsym() creates a global symbol for a message. This macro is used so that the naming convention can be changed without changing any of the references.

msgsym() takes the error message name and facility. Be sure to use the **&** operator in front of **msgsym()**.

errormsg() expects a pointer to the error message and any other fields specified by printf-like conversion characters, as returned by **msgsym()**, after “expects a pointer”.

19.5.1 Example of the **errormsg()** Function

The following **msgdef()** and **errormsg()** calls produce the “%IPX-3-TOOMANYNETS: Too many networks” error message:

```
msgdef(TOOMANYNETS, IPX, LOG_ERR, 0, "Too many networks");
errormsg(&msgsym(TOOMANYNETS, IPX));
```

19.5.2 Example of the **errormsg()** Function With Variables

The following **msgdef()** and **errormsg()** calls produce the “%AAAA-3-RELOGIN: sanity check in re-login rsmith to rjones” error message. Note the use of the **%s** **printf()** conversion character.

```
msgdef(RELOGIN, AAAA, LOG_ERR, MSG_TRACEBACK,
       "Sanity check in re-login %s to %s");

if (newuser->acctQ.qhead || newuser == olduser)
    errormsg(&msgsym(RELOGIN, AAAA), olduser->user, newuser->user);
```

Note Use %s only for names, such as interface names or router names, and to output certain constructed strings. It is an error to use %s to pass textual phrases that can vary from one invocation to the next.

19.5.3 Use errmsg_ext() in Remote Registry Calls

There is a problem with using errmsg() that happens with platforms (e.g. the Line Card platform) and systems (e.g. ION) that need to send error messages for errmsg() from an address space where the messages originated to another address space in IOS where the errmsg() function resides.

For example, when the ... variable argument list has no way to be passed over with a *remote* registry call, the current practice is to print out these arguments from the variable argument list to become one supported data type and then pass it to errmsg() to be stored in another address space. By doing so, because the message cannot use the errmsg() function, *the message cannot be filtered* by many applications that are associated with errmsg(). Those applications/filters in *remote* registry calls check the format string and look for each formatting directive, such as %s, %d, %c, etc., and expect there to be a one-to-one corresponding parameter for each directive. If any of the parameters are missing, one or more of those applications/filters will be broken and may crash.

The errmsg_ext() API function was developed to resolve this problem by taking char ** to substitute the variable argument list for the receiving message by IOS errmsg():

```
errmsg_ext(const message *, char **)
```

The errmsg_ext() API is the same as errmsg() except for the different argument type. In order to use the errmsg_ext() API, the caller to the errmsg_ext() function must print out each argument in the variable argument list and store each of them into an array of pointers to char (char **) with (char **)NULL as the last element in the array.

19.5.3.1 Adding errmsg_ext() to Remote Registry Calls

Suppose you have a remote registry named ios_errmsg and you want to invoke errmsg() in the calls to your remote registry. In this case, you can use errmsg_ext() in the your registry calls (for example, use reg_add_ios_errmsg(errmsg_ext, "errmsg_ext") and reg_invoke_ios_errmsg(errmsg_ext, "errmsg_ext")).

Here is an example using errmsg_ext() that assumes a platform-specific function named var_to_pp() mallocs memory for the buffer and translates va_list into char **:

```
my_errmsg(message *msg, ...);
{
    char **buffer;
    va_list args;
    va_start(args, msg);
    var_to_pp(buffer, len, args);
    reg_invoke_ios_errmsg(msg, buffer)
    va_end(args);
    .
    .
    .
}
```

19.6 Coding Error Messages Example

The following example is taken from the `stp/stp_msg.c` file. It illustrates the format of an `msg_xxx.c` file and the text to include in each of the file's components:

```
#include "master.h"
#define DEFINE_MESSAGES TRUE
#include "logger.h"
facdef(SPANTREE);
msgdef_section("Spanning Tree error messages");
msgdef_limit(PORT_SELF_LOOPED, SPANTREE, LOG_ERR, 0,
             "%s disabled.- received BPDU src mac (%e) "
             "same as that of interface",
             MSGDEF_LIMIT_GLACIAL);
msgdef_explanation(
    "The source MAC address contained in a BPDU that was received on the
    interface specified in the error message matches the MAC address assigned to
    that interface. This indicates that a port may be looped back to itself. This
    condition might be caused by a diagnostic cable that is being plugged in. The
    interface will be administratively shut down.");
msgdef_recommended_action(
    "Check the interface configuration and any cable that is plugged into the
    interface. Once the problem has been resolved, reenable the interface by
    entering the <CmdBold>no shutdown<NoCmdBold> command in interface
    configuration mode.");
```

In the *Cisco IOS Software System Messages* manual, the `SPANTREE PORT_SELF_LOOPED` message would appear as follows:

```
%SPANTREE-3-PORT_SELF_LOOPED: [chars] disabled.- received BPDU src mac
([enet]) same as that of interface
```

Explanation The source MAC address contained in a BPDU that was received on the interface specified in the error message matches the MAC address assigned to that interface. This indicates that a port may be looped back to itself. This condition might be caused by a diagnostic cable that is being plugged in. The interface will be administratively shut down.

Recommended Action Check the interface configuration and any cable plugged into the interface. Once the problem is resolved, reenable the interface by entering the **no shutdown** command in interface configuration mode.

Debugging and Error Logging

Added new section 20.28 “The sdec Tool and Stack Corruption Troubleshooting”. (May 2011)

Added new section 20.27.3 “ASLR and Impact on Debugging”. (September 2010)

Added new section 20.26 “Cisco Error Number (errno)”. (March 2010)

Added new section 20.27 “Debugging ASLR Enabled Cisco IOS Images”. (June 2010)

Added new section 20.5 “Enable Debug commands during boot-up in IOS” (June 2010)

20.1 Debugging and Error Logging Facilities Overview

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, and so on.

Note Cisco IOS debugging and error logging questions can be directed to the interest-os-logging@cisco.com, interest-syslog@cisco.com, and syslog-dev@cisco.com aliases.

To make certain debug processes faster, you can redirect **show** command output to a file on the router and then upload the file, as shown in the following examples (see EDCS-135330 for information on the CLI Redirection Capability):

```
show tech | redirect tftp://171.69.1.129/gst/showoutput.txt
```

```
show tech | redirect ftp://gst:MYPASSWORD@171.69.1.129//tftpboot/gst/showtech.out
```

This chapter describes the following (does not include all sections in the chapter):

- 20.2 The Debug Facility and Exceptions

This is used in the development environment to analyze a core file using GDB, using the **debug** command or using compile time conditionals. This facility can also be used in extenuating circumstances at the customer site.

- 20.8 The Event Trace Facility

This is used in the development environment and is used in debugging on a subsystem basis, identifying and storing recent events defined by the engineer.

- 20.14 What is the Enhanced Error Message Log Count

This facility is particularly useful at the customer site, although it can of course be used in the development environment. This facility changes the error message logging algorithm to retain only one entry for each error message and adding a count of the number of times issued. This helps in large production environments where many messages are issued.

- 20.15 The Receive Latency Trace Facility

This is used in the development environment as it may have significant performance impacts. It is used in analyzing Received Packets to capture the time spent processing different types of packets.

- 20.21 Traceback Recording

This is used in the development environment as it may have performance impacts. It provides a means for storing and retrieving tracebacks, giving the developer a common interface and saving coding time.

- 20.26 Cisco Error Number (errno)

The Cisco Error Number, `errno`, is a structured error return code that can be used to indicate a globally unique subsystem-specific error.

- 20.28 The sdec Tool and Stack Corruption Troubleshooting

You might be able to use the **sdec** tool created by Preston Chilcote (pchilcot) to troubleshoot stack corruption.

Note The facilities described in this chapter are used to capture completely different data.

20.1.1 Debugging and Error Logging Facilities Comparison

Table 20-1 compares the various debugging and error messaging facilities, whether they require a special image and extra coding, and whether they are appropriate for installation and use at a customer site.

Table 20-1 Comparison of Debugging and Error Logging Facilities

Facility	Special Image Required	Customer Site Use
Debugging Core Files	No	Yes
Using the debug command	No	Yes
Compile Time Debugs	Yes	No
Event Trace	No	Yes
Enhanced Error Msg Log Count	No	Yes
Receive Trace Latency	Yes	Yes
Traceback Recording	No	Yes, with a symbol file handy

20.1.2 Event Tracing versus Debugging

There is a slow trend within IOS subsystems to use the event-trace mechanism instead of debugging. The motivation behind this trend is the ability to debug startup conditions and ability to debug customer issues after the event. However, this trend can yield a performance penalty. The reasons

why and when you should do event tracing versus debugging are described in the pros and cons of event tracing and the pros and cons of debugging in Table 20-2 and Table 20-3, followed by information on how to minimize the costs of event tracing and how to improve event tracing.

Table 20-2 PROs and CONs of Event Tracing

Event Tracing PROs	Event Tracing CONs
Shows conditions at startup and preceding an event.	Additional overhead caused by event logs being enabled.
It is most helpful to know what happens leading up to an event in order to figure out the trigger versus what it looks like after it is broken. Event tracing has the ability to debug startup conditions before the CLI allows debugs to be enabled. That is one of the main failures of debugs; they are never on when the problem happens the first time. Debugs are good for things like “authentication failures for users” when they can be turned on conditionally (that is, per user, per interface, etc.).	Each logged event costs something, but by logging only raw unformatted data and formatting only when the event log is shown, this overhead is minimal. Also, it is recommended that if a large chunk of memory is needed, implement a basic event trace that is always on and that has some information about what the system is doing for the particular feature. Then, implement a more detailed event trace that is off by default and is configurable.
Event tracing has proven extremely useful for debugging problems seen in customer networks because you can see the sequence of events logged when the customer experienced problems, something debugs cannot do in cases where the problem is not easily reproducible.	A good idea is a hierarchical event trace implementation for the subsystem, but yet try to predict some level of tracing that is on by default to give information leading up to a problem in a production network.
Event traces using the event-trace infrastructure also give the ability to merge traces from different subsystems to allow for the correlation of events when trying to debug a problem.	Also, you can configure monitor event-trace component continuous , which effectively gives you live debug.
Event tracing does not require customers to enable debugs, something customers often are reluctant to do on production routers.	Therefore, you can remove some debug <code>buginf()</code> code when converting over to use the event-trace facility.
Easy to use.	Could be more “non-coder readable.”
Event tracing is somewhat easy to use if you know what you are looking for. For example, one common logging API across <i>all</i> components, with one “zero” for timestamps that allows comparisons between different component logs. This is easier to use than multiple different logs, all with different configuration / show commands.	The CEF team was one of the first teams to use the common event trace infrastructure and it has been a huge success, although it could be a little more “non-coder readable.”
Event trace allows you to get the wanted data in tricky situations.	

Table 20-2 PROs and CONs of Event Tracing (continued)

Event Tracing PROs (continued)	Event Tracing CONs (continued)
Can minimize processor load with sensible usage.	Hidden performance penalties.
Regarding valid concerns about the hidden processor load, it is best to educate users of this facility about what operations are most expensive, so that sensible use can be made of event trace (disabling it is not the way to go).	The performance penalty of the event-tracing is hard to gauge, it is entirely hidden by the caller. Some developers believe that all event tracing should be disabled in production images, and only enabled from the config on a as-needed basis.
We cannot disable event-trace because we get major complaints when a customer has an outage and we cannot give them a root cause analysis because we do not have any data leading up to the event.	
In the Cat6k EARL8 for example, we have specifically asked for “critical event” tracing as a requirement (see ENG-504609 for details), which will be logged into Systems Event Archive (SEA).	
At least in the high-end platforms, customers are asking for auto-debuggability and manageability and they do not mind paying the extra CPU cycles. They would much prefer us get to the root cause of the problem the first time, every time. This is what we have heard in all customer EBCs and TABs.	
Event tracing should not be globally disabled in production images and require config to enable them, because this guarantees that when the catastrophic once-in-a-blue-moon bug happens at a customer site, the relevant event log will be off.	
Can be used “live” to monitor problems.	Continuous event trace impacts performance.
Can be used to do a “live” debug by the monitor event-trace component continuous command.	Depending on the circumstances, performance could take priority over event tracing.
There is also the benefit that when timestamps across all event trace logs in multiple components have the same start time, you can compare or interweave them to see what has happened across multiple components. You can switch the event-trace mechanism to “continuous” so it acts just like debug commands.	
String manipulation can be avoided.	String manipulation causes performance penalties.
What string manipulation? The event logging dumps raw data, not strings, into the event log buffer. Only when the logs are printed is the data converted to strings. If event log users are printing strings into the event log buffers, then we need to fix the event log users.	Event tracing has the ability to debug startup conditions and customer issues after the event with a potential performance penalty. Though the event-trace mechanism appears to be quite efficient, a significant amount of string manipulation is going on at all times in the background.

Table 20-2 PROs and CONs of Event Tracing (continued)

Event Tracing PROs (continued)	Event Tracing CONs (continued)
Educate to prevent performance impact. List out the parts that impact performance. Inform users; just because a feature <i>may</i> get abused does not mean that you should switch it off for everyone. Also, the event-trace component owners can run a regular audit to keep the code healthy. You need to make sure default sizes are sensible and that stacktrace is only enabled in rare cases by default. CEF trace sizes are suitably reduced in size for the 3750 platform to avoid memory problems. The recommended use of the event logging infrastructure is to log raw data. As an example, the CEF table trace event structure looks like this:	No guarantee that performance will not be impacted. Event tracing can impact performance if you are not careful. How do you guarantee that performance will not be impacted?
<pre data-bbox="143 777 719 1389">typedef struct fib_table_event_record_ { union { struct { ipaddrtype address; ipaddrtype mask; } ipv4; struct { in6_addr_t address; uchar mask_bits; } ipv6; } addr; fib_af af; tableid_t tableid; fib_trace_event_od od; FIB_TABLE_EVENT_TYPE type; uchar subcode; uchar result; db_epoch epoch; uchar epoch_unknown; ulong elapsed_time; } fib_table_event_record;</pre>	
No strings, although we do store an IPv4 or IPv6 address (which is perhaps quite a large record, but you need to log this somehow).	
Provides history with debugs. Tells a history of events opposite debugs that were not captured unless they were turned on.	Enabling traceback generation by default is costly. One hidden cost is the traceback generation. Great care should be taken by event-trace users when enabling traceback generation by default.
Can be dumped to a file. Can be dumped to a (remote) file via tftp .	

Table 20-2 PROs and CONs of Event Tracing (continued)

Event Tracing PROs (continued)	Event Tracing CONs (continued)
Component logger code not needed.	Potential code bloat.
As opposed to each component implementing its own logger code? Surely the reverse - it will reduce code size.	If every component adds event tracing, there is potential code bloat. Balance this against each component <i>not</i> having its own logger code. CEF used to have its own logger code, but that has been removed. This should save memory, and will certainly reduce code maintenance requirements by using common code.
Memory is not a problem for mid/high-end platforms.	Memory is a problem for low-end platforms.
Memory is a concern for the low-end platforms. However, this should not be a problem for other platforms (such as Cat6k). Additionally for Cat6k, with Systems Event Archive (SEA), you can archive events into huge Compact Flash, and then the need to use main memory will be very minimal.	There is potential memory usage. On the truly low-end boxes (the Catalyst 2k and 3k series switches, for example) you will not get lots more memory and CPU any time soon, and you will never get enough memory and CPU on this kind of box that you can afford to be careless with it.
We are learning a lesson about memory. The new ISRs have 1G and customers will pay for the added debug capability at the expense of memory when we prove to them we can solve our own bugs. Nothing hurts them more than having to try to debug something in a production network that is intrusive.	The CEF people have provided a way to size their event log buffers differently for different platforms, which seems to work well for now on the Catalyst 3750. It's worth noting that we make the buffer sizes quite a lot smaller than the FIB component originally asked for: for example, the FIB trace buffer holds 10,000 entries by default, but we size it so that it holds only about 1,000 entries on the Cat3750. Other components go along with the FIB (fibif, adj, ipv4fib, ipv6fib, xdr, fibrp, fiblc) whose buffers we also make a lot smaller than default. We limit the memory usage for all these components together to about 650 kbytes.
	If we had 20 additional features that each wanted buffers for multiple components, we would start to worry about the amount of memory being set aside just for tracing events, particularly on the SKUs that have only 128 MB or less. So the existing solution's memory usage does not scale well to support large numbers of components each doing its own event tracing.
	And 20 additional features that asked for default buffer sizes as large as the various FIB component defaults with no way for the platform to override the allocations would simply be unsupportable on our platforms. (Take this as a plea to the component owners to provide a way for the event buffer to be different-sized on different platforms).
Memory usage can be limited.	Memory usage tolls.
The recommendation is to only allocate the event log buffer when the feature is configured, and not do it at startup or subsystem init. Limit the buffer size to something reasonable and event trace should not be on unless your feature is enabled.	If every component dumps tons of events, at some point it will be a bottle neck, especially on highly scaled boxes. The elog is almost performance hit free, but that mechanism is different than what CEF/FIB is using.
For more information, see section 20.8, "The Event Trace Facility," Functional Spec EDCS-84509, and Design Spec EDCS-95250.	
Also, you can make the memory allocation platform-specific (via <code>platform_get_value()</code>), related to the number of interfaces, or whatever makes sense.	

Table 20-2 PROs and CONs of Event Tracing (continued)

Event Tracing PROs (continued)	Event Tracing CONs (continued)
<p>Common infrastructure can use less memory. CEF uses event tracing instead of the old CEF-specific logging code. It may be that CEF now uses <i>less</i> memory by using this common infrastructure.</p> <p>Using event trace in multiple components, you get rid of each component's own logger code, so at least it is only one set of code to be made more efficient.</p>	<p>CPUs need to be more powerful for event trace. There is an absence of powerful CPUs to do such useful stuff. The trace system starts out useful when a few clients are using it, but as a large number of clients start using it, the impact is significant and drags the CPU down quietly in the background. So, while we like the trace idea 100%, we have to start thinking about how to build more memory and CPU into boxes in anticipation of more clients using it.</p>

Table 20-3 PROs and CONs of Debugging

Debugging PROs	Debugging CONs
<p>Lower impact on performance than event tracing. Compared to the performance penalty for event tracing on low-end platforms, debugging does not impact performance as much.</p> <p>However, any debug code impacts performance. For example:</p> <pre data-bbox="143 952 442 1100">if (debug_flag_set) { do_something(); }</pre> <p>checks the flag, even if not debugging. It is less than putting in event-trace code, but it is not free. If we littered the CEF forwarding path with such code, it would be very slow. We have a different approach to debugging there, as we build output chains of elements that represent operations on the packets, and only if debug is enabled do we add a “debug” element, thus making the impact of disabled debug precisely zero.</p>	<p>Performance penalty. Even unused debugs have hurt performance on the low end. For instance, the simple check “if (debug) then print” is costly, even if debug is not on. This is usually on systems that are experiencing cache misses. The extra debug checks and chopped up code paths can add to cache thrash. Nothing is free, especially when your CPU is being pushed beyond its limits, as has happened on the low-end fastpath in recent years.</p>
<p>Allows research that is not available otherwise. Debugging can help you to do research that cannot be done via event tracing; for example, debug is useful when you need a lot of specific information that would be too costly (memory and/or performance) to store in an event log. Often, you know that getting the debug info will affect performance, but this is acceptable as it is a short term hit. There is almost always a very small hit when debug code is present but not enabled via the CLI, but this is probably less than the event trace code.</p> <p>In general, event tracing can get you to the right area of code, and then you can switch to focused debug info if the event tracing does not provide enough detail.</p>	<p>Not used in production networks. Customers will simply not turn on debugs in production networks anymore. They have been burnt one too many times by it taking the box down usually because they did not turn off logging to the console.</p> <p>Customers in high profile networks will NOT turn on debugs. Try asking one of the financial firms to enable a debug on their production network.</p>

20.1.2.1 How to Balance the Benefits of Event Tracing with the Costs of Event Tracing per Platform

Event trace is unique in providing prefault context information, which is especially valuable when a problem is difficult to duplicate or a customer does not want to turn on the debug facility on their live network. This information can speed up root-cause discovery and help us deliver a solution sooner, which makes customers happier with us.

The question is usually not whether to use event tracing, or whether event tracing should always be turned off by default. The question is how to balance the benefits with the costs, which is a per-platform decision.

Here are some ways for minimizing cost:

- When development is complete, scrub out event trace messages that were added to assist in development and internal testing. Ensure that each event trace is nonredundant and adds significant value in a post-FCS environment, or pull the event trace out before commit. Toss out the obsolete junk.
- If space is tight, compress. Where practical, use literal strings in the text segment defined at compile time instead of verbose strings copied/built at run-time. All that is really needed to identify a code point is a few letters and a unique number. One area this can be done in easily is exception conditions like executing down unexpected code paths. For example, “TO:AE35” communicates to anyone with source access that there was a timeout and its exact location in the code just as effectively as “Timeout in ATA disk subsystem: file abc_xys: line 230” does. But the former, “TO:AE35”, can fit in a literal in the .c file, and you can put an extern in the header file. The latter might have been built at run-time.
- Abbreviate messages full of parameters.
- For nonexceptional messages like state transitions, aggressively using abbreviations can at least reduce memory consumption, if not improve speed.
- Human-readable output is faster to follow and analyze, so it pays not to go too far in search of optimal efficiency unless the platform limitations demand this. A balance needs to struck between code+memory efficiency versus troubleshooting speed. That is a code+per-platform decision. Code shared between platforms ought to be treated as the lowest platform, of course. So that it is easy to make the reconstituted log human-readable while storing using optimal efficiency methods. Store string pointers/integer-ids, and convert to text later. Use of .enum files with the option to have a name (LONG/SHORT) associated with each enum guarantees they will match up.
- Each platform might need its own thresholds and guidelines for when an event trace message is justified. Even a heavily compressed set of event messages beats out nothing at all when faced with a total lack of information from the field.

20.1.2.2 Possible Improvements for Event Tracing

Instead of using string manipulation, you can store opcodes and raw data. The opcodes can be converted to human readable text either on the router or with a “host-based” tool. Opcode-based debug will also allow for more state to be stored.

You *should* wherever possible use opcodes and raw data. Only store strings if absolutely necessary. An example of the latter is the CEF interface logs where (due to `ldb/ifindex` reuse) we have to store the short namestring to avoid printing out bogus interface names later where an `ifindex` value has been reused.

To intelligently assess the value of encoding in a machine-only readable format on a given platform depends on the actual percentage of CPU resources and memory that are consumed by the event-trace facility.

Given that event-trace logs can be quite long, and speed is likely of the essence if you're event tracing on a customer's router, the decode function to convert opcodes to human-readable format should be on the router. For example, the CEF table event log uses this structure of raw/opcode data:

```
typedef struct fib_table_event_record_ {
    union {
        struct {
            ipaddrtype address;
            ipaddrtype mask;
        } ipv4;
        struct {
            in6_addr_t address;
            uchar      mask_bits;
        } ipv6;
    } addr;
    fib_af          af;
    tableid_t       tableid;
    fib_trace_event_od od;
    FIB_TABLE_EVENT_TYPE type;
    uchar           subcode;
    uchar           result;
    db_epoch        epoch;
    uchar           epoch_unknown;
    ulong           elapsed_time;
} fib_table_event_record;
```

And, uses this function to format/decode the opcodes:

```
/*
 * fib_table_trace_pretty_print
 */
boolean
fib_table_trace_pretty_print (fib_table_event_record *event, char *prefix,
                             char *p_output)
{
    fib_table *table;
    char *e_str, *extra_str;
    const char *red_mode_str;
    int e_str_spc, e_str_inc, e_body_len, e_prefix_len;

    /*
     * Setup stuff for sprintf() calls
     */
    e_str = p_output;
    e_str_spc = EVENT_TRACE_MAX_PRINT_SIZE;

    /*
     * Find the FIB control block.
     */
    if (event->tableid != FIB_TABLE_NULL_ID) {
        table = fib_table_find_internal(event->tableid, event->af);
    } else {
        table = NULL;
    }
```

```

/*
 * Print table
 */
if (event->tableid == FIB_TABLE_NULL_ID) {
    e_str_inc = 0;
} else {
    e_str_inc = strlen(table ? table->name : "?xxx?") + 2;
}
/*
 * printf("%*c", 0, ' ') will insert one space, inserting an unneeded
 * space for tables with 10 char names.
 */
if (e_str_inc < 10) {
    e_str_inc = snprintf(e_str, e_str_spc, "%*c", 10-e_str_inc, ' ');
    e_str += e_str_inc; e_str_spc -= e_str_inc;
}
if (event->tableid != FIB_TABLE_NULL_ID) {
    if (table) {
        e_str_inc = snprintf(e_str, e_str_spc, "[%s]", table->name);
    } else {
        e_str_inc = snprintf(e_str, e_str_spc, "[?%03d?]",
event->tableid);
    }
    e_str += e_str_inc; e_str_spc -= e_str_inc;
}

/*
 * Print address
 */
e_str_inc = snprintf(e_str, e_str_spc, " %s", prefix);
e_str += e_str_inc; e_str_spc -= e_str_inc;
e_prefix_len = e_str_inc;

/*
 * Print epoch
 */
if (!event->epoch_unknown) {
    e_str_inc = snprintf(e_str, e_str_spc, "'%02x", event->epoch);
    e_str += e_str_inc; e_str_spc -= e_str_inc;
    e_prefix_len += e_str_inc;
}

/*
 * Pad to event body
 */
e_str_inc = snprintf(e_str, e_str_spc, "%*c", 23-e_prefix_len, ' ');
e_str += e_str_inc; e_str_spc -= e_str_inc;

/*
 * Print fixed event data
 */
e_str_inc = snprintf(e_str, e_str_spc, "%s",
FIB_TABLE_EVENT_TYPE_get_string(event->type));
e_str += e_str_inc; e_str_spc -= e_str_inc;
e_body_len = e_str_inc;

```

```

/*
 * Event body
 */
e_str_inc = 0;
switch (event->type) {
case FIB_TABLE_EVENT_RIB_UPDATE:
    e_str_inc = snprintf(e_str, e_str_spc, " %s",
                         fib_rib_update_type_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_FIB_INS:
    break;
case FIB_TABLE_EVENT_FIB_REM:
    e_str_inc = snprintf(e_str, e_str_spc, " %s",
                         FIB_TABLE_EVENT_SC_RE_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_FIB_MOD:
    FIB_TABLE_EVENT_SC_M_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_FIB_ADD_SRC:
    e_str_inc = snprintf(e_str, e_str_spc, " %s%s",
                         fib_fib_src_get_string_short(event->od.u.uint.other_data),
                         FIB_TABLE_EVENT_SC_ADD_SRC_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_FIB_REM_SRC:
    e_str_inc = snprintf(e_str, e_str_spc, " %s%s",
                         fib_fib_src_get_string_short(event->od.u.uint.other_data),
                         FIB_TABLE_EVENT_SC_RE_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_FIB_MOD_SRC:
    e_str_inc = snprintf(e_str, e_str_spc, " %s",
                         fib_fib_src_get_string_short(event->od.u.uint.other_data));
    break;
case FIB_TABLE_EVENT_RECEIVE:
    e_str_inc = snprintf(e_str, e_str_spc, " %s",
                         FIB_TABLE_EVENT_SC_R_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_BROKER:
    e_str_inc = snprintf(e_str, e_str_spc, " %s %s",
                         fib_table_record_actions_get_string(event->subcode),
                         FIB_TABLE_EVENT_SC_B_get_string(event->od.u.uint.other_data));
    break;
case FIB_TABLE_EVENT_DEFAULT:
    e_str_inc = snprintf(e_str, e_str_spc, " %s",
                         FIB_TABLE_EVENT_SC_D_get_string(event->subcode));
    break;
case FIB_TABLE_EVENT_GLOBAL:
    e_str_inc = snprintf(e_str, e_str_spc, "%s",
                         FIB_TABLE_EVENT_SC_G_get_string(event->subcode));
    e_str += e_str_inc; e_str_spc -= e_str_inc;
    e_body_len += e_str_inc;
/*
 * Some of the global events have additional information to print.
 */
switch (event->subcode) {
case TABLE_EVENT_SC_FLUSH_TABLE:
    e_str_inc = snprintf(e_str, e_str_spc, " (%d/%dms)",
                         event->od.u.uint.other_data,
                         event->elapsed_time);
    break;
}

```

```

case TABLE_EVENT_SC_ADD_TABLE_SOURCE:
    /* fall through */
case TABLE_EVENT_SC_REM_TABLE_SOURCE:
    e_str_inc = sprintf(e_str, e_str_spc, " %s",
                        (char *)event->od.u.uint.other_data);
    break;
case TABLE_EVENT_SC_FIND_TABLE:
    /* fall through */
case TABLE_EVENT_SC_FIND_CREATE_TABLE:
    e_str_inc = sprintf(e_str, e_str_spc, " %d", event->tableid);
    break;
case TABLE_EVENT_SC_PURGE_FIB_DONE:
    e_str_inc = sprintf(e_str, e_str_spc, " (%d)",
                        event->od.u.uint.other_data);
    break;
case TABLE_EVENT_SC_SWITCHOVER_START:
    /* fall through */
case TABLE_EVENT_SC QUIESCE_START:
    red_mode_str = reg_invoke_red_mode_convert_mode_to_string(
        event->od.u.uint.other_data);
    e_str_inc = sprintf(e_str, e_str_spc, " %s(%d)",
                        red_mode_str ? red_mode_str : "unknown",
                        event->od.u.uint.other_data);
    break;
case TABLE_EVENT_SC_BRK_PURGE_EPOCH:
    /* fall through */
case TABLE_EVENT_SC_BRK_NEW_EPOCH:
    e_str_inc = sprintf(e_str, e_str_spc, " %s",
                        FIB_TABLE_EVENT_SC_B_get_string(event->od.u.uint.other_data));
    break;
case TABLE_EVENT_SC_NEW_RIB_RATE_PEAK:
    e_str_inc = sprintf(e_str, e_str_spc, " %d",
                        event->od.u.uint.other_data);
    break;
default:
    e_str_inc = 0;
}
break;
case FIB_TABLE_EVENT_XDR:
    e_str_inc = sprintf(e_str, e_str_spc, " %s",
                        fib_table_record_actions_get_string(event->subcode));
    break;
}
e_str += e_str_inc; e_str_spc -= e_str_inc;
e_body_len += e_str_inc;

/*
 * Allow register users to add any addition printing
 */
extra_str = string_getnext();
if (reg_invoke_fib_table_trace_other_data(
    event->type, (char *)&event->od, extra_str)) {
    e_str_inc = sprintf(e_str, e_str_spc, " %s", extra_str);
    e_str += e_str_inc; e_str_spc -= e_str_inc;
    e_body_len += e_str_inc;
}

```

```

/*
 * Pad to event status
 */
e_str_inc = snprintf(e_str, e_str_spc, "%*c", 25-e_body_len, ' ');
e_str += e_str_inc; e_str_spc -= e_str_inc;

/*
 * Event status
 */
e_str_inc = snprintf(e_str, e_str_spc, "%s",
                     TABLE_EVENT_RC_get_string(event->result));
e_str += e_str_inc; e_str_spc -= e_str_inc;

return (TRUE);
}

```

Which produces output like this:

00:00:05.068:	[VRF00065]	10.71.8.168/30'00	FIB add src RIB (ins)	[OK]
00:00:05.068:	[VRF00065]	10.71.8.169/32'00	FIB add src I/F (ins)	[OK]
00:00:05.068:	[VRF00065]	10.71.8.168/32'00	FIB add src I/F (ins)	[OK]
00:00:05.068:	[VRF00065]	10.71.8.171/32'00	FIB add src I/F (ins)	[OK]
00:00:05.068:	[VRF00065]	10.71.8.170/32'00	FIB add src RIB (ins)	[OK]
00:00:09.188:	[Default]	195.249.246.101/32'00	FIB add src Adj (ins)	[OK]
00:00:09.348:	[Default]	195.249.9.41/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.61.238/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.61.240/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.61.237/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.0.18/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.61.243/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.61.243/32'00	FIB mod src RIB	
	[Ignr]			
00:00:09.348:	[Default]	195.249.0.102/32'00	FIB add src RIB (ins)	[OK]
00:00:09.348:	[Default]	195.249.0.102/32'00	FIB mod src RIB	
	[Ignr]			
00:00:09.372:	[Default]	195.249.246.103/32'00	FIB add src Adj (ins)	[OK]
00:00:09.472:	[Default]	195.249.246.102/32'00	FIB add src Adj (ins)	[OK]

Make the output in the event traces as dummy-proof as possible so that customers (as well as TAC) can make an attempt at using them to solve their own problems, or else the customers will simply end up with TAC cases and emails back to DE just to help them interpret the event logs. Event logs should help anyone working on the box, not just a developer fixing a bug.

The function needs to be written, and then modified with every new message added. Furthermore, the function needs to be kept in sync with every branch of IOS or IOX. So there could be a nontrivial added maintenance cost associated with the memory and CPU cycle savings. That has to be weighed against the resource costs of longer, human-readable strings.

In the absence of hard data, human-readable strings should stay except for platforms where memory or CPU cycles are seriously squeezed, and where the event-trace facility is a nontrivial contributor. The thing is, without hard data, it is not evident which platforms these would be.

CEF is handling this by writing CEF tracing data in raw form (opcodes) into the event tracing, so there is little overhead involved. Then, upon viewing the event-trace, it is translated into human-readable form. Therefore, the string manipulation CPU hit is only encountered if the data is viewed, and at the time of the operators' choosing.

The HA subsystems that encode the strings up front are planned to be converted to the CEF design where it is only formatted at viewing time.

20.2 The Debug Facility and 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.

Note Only syslog and error messages of severity level equal or higher than `log_reload` (`log_reload` holds severity level of logging during reload) are logged when you issue `reload` on the router. There is a limit on the maximum number (`max_log_messages_at_reload`) of syslog and/or error messages that can be logged once reload has started. The limit is configurable. The CLI to configure `log_reload` and `max_log_messages_at_reload` - [no] **logging reload** [`message-limit number`] [`emergencies` | `alerts` | `critical` | `errors` | `warnings` | `notifications` | `informational` | `debugging` | <0-7>]

20.2.1 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.

One useful tool for troubleshooting crashes is the **debug sanity** command. Unless you are doing performance/stress/scalability testing, you can enable **debug sanity** always so that if a corruption in packet memory occurs you can catch that immediately with a crash. How do you know that **debug sanity** has found a problem? The traceback usually points to `retbuffer/getbuffer`.

For example:

```
<snip>
0x6093BA1C:abort(0x6093ba14)+0x8
0x60939728:crashdump(0x60939610)+0x118
0x6096D850:validmem_complete_interrupt(0x6096d794)+0xbc
0x6089B004:pak_pool_item_validate(0x6089afb0)+0x54
0x60896E54:retbuffer(0x60896d0c)+0x148
</snip>
```

Will **debug sanity** put a message in the crashinfo file? In crashinfo after “===== Register Memory Dump =====”, you can see buffer check=1 if **debug sanity** is enabled. Otherwise it would be 0.

For example:

```
<snip>
ios109#more slot0:crashinfo_20051121-152513 | in buffer check
buffer check=1 sched_hc=0x0
</snip>
```

DEs usually ask to reproduce a packet memory corruption problem with **debug sanity** enabled if the crashinfo does not seem to provide accurate information. **checkheaps**, a low priority process in Cisco IOS that runs by default, also detects corruptions and crashes the router, but by the time this process catches the corrupted packet, the corrupted packet would have passed through many other processes.

If **debug sanity** is enabled, it catches corruption immediately before the corrupted packet is returned/obtained from the pool. From a testing point of view, **debug sanity** avoids a second cycle of reproducing a problem and has little overhead.

debug sanity does not spew any messages to the console. If any corruption is noticed, **debug sanity** performs extra validations for a packet and crashes the router. **debug sanity** also performs additional validation to the elements that are enqueued/dequeued in various queue structures and crashes the router if any anomaly is seen. You need to enable **debug sanity** when you are suspicious of memory corruption and want to crash the router immediately to get useful information for debugging. For more information on **debug sanity**, see the following URL:

<http://gabbar.cisco.com/shiv/osinfra/faqs.htm#B9>

In Release 12.0, the tools for troubleshooting crashes 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.

Note As of Release 12.2S and 12.3, the core files generated from IOS will now carry a timestamp of the form “_YYYYMMDD-HHMMSS.” For example, `core_19950601-132345.z`

20.2.1.1 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 1 Specify the transfer protocol. If you omit this command, TFTP is used.	exception protocol {ftp rcp tcp}
Step 2 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 3 Specify the name of the core file. If you omit this command, the core file is named <i>routernamename-core</i> , where <i>routernamename</i> is the name of the router set with the hostname command.	exception core-file <i>dump_filename</i>
Step 4 Write the core file.	exception dump <i>dump_host</i>

20.2.1.2 Analyze a Core File

You can analyze a core file with GDB, or UNIX tools.

Analyze a Core File with GDB

The Cisco GDB (the Cisco GDB is an extension of the GNU/FSF GDB) debugger contains support for reading Cisco IOS core files. For information about using GDB, see the *Cisco Engineering Tools Guide*.

<http://wwwin-enged.cisco.com/common/doc/tools/>

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.

20.2.2 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 in Table 20-4.

Table 20-4 ROM Monitor Debugging Command

Command	Explanation
alter	Changes and examines memory
stack	Provides a traceback

20.2.3 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.

At the time of writing, the current version of GDB used for IOS development is GDB 6.1.

For information about GDB, see:

<http://www.gnu.org/software/gdb/>

Here is a pointer to the GDB FAQ:

http://wwwin-enged.cisco.com/ios/courses/gdb_faq/GDB_FAQ1.htm

The EDCS documents for GCC 3.4/GDB6.1 are:

EDCS-383144—a PRD with all the new features from FSF/GNU (additions from GDB 5 that GDB 6 has.)

EDCS-383145—the gdb6.1 functional spec. It's a recap of the new features that were added to GDB 6 (above and beyond the FSF/GNU features), which were later ported to GDB 6.

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.

20.2.3.1 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.

For example, type:

#g k

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.

20.2.3.2 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 as 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.

20.3 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.

If a significantly large number of messages are generated such that the console output is flushed, the messages may appear out of order. If the logging buffer is enabled, the output of `show logging` shows the messages in the proper sequence.

Tracebacks can be enabled for all debug messages (and all error messages) with this command, and this command (or calls to `bugtrace()`) can be used to show stack traces.

20.3.1 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 `CONTTY`.

20.3.1.1 Differences between `buginf()` and `printf()`

`buginf()` should be used only in conjunction with an enabled debug flag that can be turned on/off via debug CLI. `printf()` should not be used in the switching path. Furthermore, `printf()` cannot be used when interrupts are disabled.

`buginf()` writes its message into a buffer that is output later. The write operation is atomic, and therefore is interrupt-safe. Furthermore, `buginf()` does not suspend. `printf()` uses assorted buffers in between the API and the actual hardware. Its big problem is that it can *block* if those buffers get full. Also `printf()` uses `forkx->tty`, so if used in interrupt context (even if that would work), it can write to random terminals/vtys/etc. `printf()` should never be used in a critical section since it may relinquish the process it is running in.

`buginf()` is output using the logger code. `buginf()` output can be directed to any of the same destinations that log output goes to. `printf()` goes to the process's tty (or to the console if the process has no tty); there is no other choice. `printf()` should never be used except in response to user input, or for certain boot-time messages that must be output to the console before the logger is running.

From a usability point of view, `printf()` is more commonly used in response to user input (e.g. in EXEC mode) and `buginf()` is used to output diagnostic messages. When adding a `buginf()`/`bugtrace()` call, consideration should be given to whether the information is important enough to be provided via an `errmsg()` call instead (meaning reported even when not debugging messages for the component are not enabled); reporting problems via `errmsg()` instead of `buginf()` can allow the customer to identify and correct a problem without contacting TAC for assistance, since the error message and recovery action can be documented in the Error Messages manual.

Note While `printf()` is used for responses to user commands and `errmsg()` is used to report errors or events that may be of interest to the user, `buginf()` and `bugtrace()` are used only for output controlled by "debug" commands. `printf()` output goes to the current process's tty; `errmsg()` and `buginf()`/`bugtrace()` output go to the configured logging destinations (console, internal logging buffer, `syslog`, and/or terminals).

20.4 How to Stop Buffered Debugging from Dropping buginf()

A number of factors can cause debugs to be lost from the logging buffer:

- Attempting to dump the logging buffer while debugs are being collected. (Solution: turn off debugs before dumping buffer.)
- The logging buffer itself is too small. (Solution: use the **logging buffered** command to configure a larger buffer.)

The logging process queue overflows because debugs are generated faster than they can be processed and logged. (Solution: configure **no logging queue-limit** to prevent the queue from overflowing.)

20.5 Enable Debug commands during boot-up in IOS

Perform the following steps to enable debug commands during boot-up:

- 1 Copy the startup-config to a tftp location.
- 2 Enter the appropriate debug command. For example, **privilege exec level 1 debug ipc packets do debug ipc packets**
- 3 Copy the file to startup-config and reload the router. The debugs will be on when the router reloads.

20.6 Debug Using Compile-Time Conditionals or Code Features

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.

There are also features created in response to defect reports that help to identify or recover from code errors that are available on some platforms or at certain release levels that often can be integrated into other release chains as needed. Links to some of these features, such as the Garbage Detector/memory leak detector tool, are provided in section 20.7, "Links to Other Debugging Documentation".

This section discusses ways to use compile-time conditionals and compiled-in code features to:

- Trace/detect packet buffer memory leaks.
- Allow an interface to recover from being "wedged" due to a suspected packet buffer memory leak.

20.6.1 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 determine 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.

20.6.1.1 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_srcaddr = 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, atalk_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,
llc2_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}, encapsc = {'\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_status = 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 = 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, 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}, encapsc = {'\000' <repeats 38 times>,
"\002\001\0046F\000\000\f\001\0028\b\000"}}, version = 4, ihl = 5, tos = 0,
t1 = 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.

20.6.2 Unwedge an Input Queue Throttled Due to Buffer Leaks

(Available in Release 12.4(6)T:) The “Input Queue Unwedging” feature described in EDCS-425693 was created to help detect at runtime and recover from a situation where an interface is wedged because it was throttled due to an out-of-buffers condition caused by a suspected packet buffer memory leak. In the case of a buffer leak, the interface will remain throttled because the out-of-buffers condition will not resolve itself over time. This section describes how to use this feature, which essentially combines operations from other features to allow the interface to be unwedged so it can continue to operate until a fix for the leak can be implemented.

The interface unwedging feature accomplishes the following:

- Detects whether packet buffer memory is taking up a certain percentage of in-use memory, which suggests a buffer leak is present.
- Identifies packet buffers that are likely to have been “leaked”.
- Uses policy maps to drop those “leaked” packet buffers to release them, and thereby trigger the interface to unthrottle.

Warning This feature should be used with extreme caution. It does not identify the code responsible for the leak, nor does it properly or safely fix the leak. It uses reliable statistics to guess whether a packet has been leaked, but does not have any way to be absolutely sure it is safe to release it. If the guess is wrong, other more serious memory corruption problems might occur.

20.6.2.1 Other Cisco IOS Features Used by the Interface Unwedging Feature

Table 20-5 describes the IOS features required by this feature in the image, and summarizes how they are used together to accomplish input queue unwedging, with indicated references for background and more information about each feature.

Table 20-5 Other Features Used by the Interface Unwedging Feature

IOS Feature/Commands	Used By This Feature To	Reference Documents
Garbage Detection in IOS, also referred to as Memory Leak Detector show memory debug leaks ... (to display suspected leaks)	Identify and deallocate suspected leaked packet buffers.	EDCS 293190: Software Functional Specification EDCS 304222: Software Design Specification. Product Feature Guide: “Memory Leak Detector”: http://www.cisco.com/en/US/products/ps6350/products_configuration_guide_chapter09186a008043fd7a.html
Memory Resource Owner Interface resource policy policy ... global memory io critical rising ... user global ...	Trigger the Garbage Detector to deallocate unwedged packets when packet memory makes up $\geq 75\%$ of the configured threshold of total in-use memory.	EDCS 298315: Functional Specification Product Feature Guide, “Interface Input Queue Memory Reclamation”: http://www.cisco.com/en/US/products/ps6441/products_feature_guide09186a0080616496.html ERM Product Feature Guide: “Embedded Resource Manager”: http://www.cisco.com/en/US/products/sw/iosswrel/ps5207/products_feature_guide09186a00803790a7.html

Table 20-5 Other Features Used by the Interface Unwedging Feature (continued)

IOS Feature/Commands	Used By This Feature To	Reference Documents
Flexible Packet Matching (FPM) class-map type access-control ... policy-map type access-control ...	Create access control policy map filters that identify and drops the packets that might be ones that were leaked.	EDCS 375882: Design Document EDCS 367690: Software Functional Specification Product Feature Guide: “Flexible Packet Matching”: http://www.cisco.com/en/US/products/ps6441/products_feature_guide09186a00805138d3.html

20.6.2.2 memory debug leak(s) reclaim Command

To allow an interface to be unwedged, this feature implements the following service internal configuration commands, as follows:

- To unwedge the interface each time it becomes wedged, use the following command in Interface Config mode:
memory debug leaks reclaim filter *filter-name* [packet-age *secs*] deallocate [exclusive]
- For a single unwedge operation on an interface, use the following command in User EXEC mode:
memory debug leak interface *interface* reclaim filter *filter-name* [packet-age *secs*] deallocate [exclusive]

Table 20-6 defines the keywords and input parameters for these commands.

Table 20-6 memory debug leak(s) reclaim Command Keywords

Keyword	Parameter	Description
interface	<i>interface</i>	Specifies the interface name to which to apply the User EXEC mode form of the command.
filter	<i>filter-name</i>	The name of the policy map to be used as a filter. Packets that match the filter constraints are those that will be dropped and released.
packet-age	<i>secs</i>	(Optional) The age in seconds after which a packet is dropped and released.
deallocate	None	Specifies to deallocate the packets that match the filter and age constraints, assuming those are the cause of the interface being wedged.
exclusive	None	(Optional) Specifies to have the interface unwedging code directly release the packets that have been dropped. If this keyword is not specified, the feature uses the Garbage Detector to release the dropped packets, which more specifically releases only suspected leaked packets. Dropped packets are those that filter <i>filter-name</i> has caught that are older than packet-age <i>secs</i> seconds.

The following configuration example shows how to use this command in Config mode to continually reclaim leaked buffers for the interface named `Ethernet3/0` using the policy map filter named `P1` for input packets that have been queued longer than 180 seconds:

```
interface Ethernet3/0
    memory debug leaks reclaim filter P1 packet-age 180 deallocate
```

The following is an example of the User Exec mode command for a single buffer leak reclaim operation on the interface named `fastEthernet 0/0` using policy map `P1`, that releases leaked packet buffers itself rather than triggering the Garbage Detector to release them:

```
memory debug leak interface fastEthernet 0/0 reclaim filter P1 deallocate exclusive
```

20.6.2.3 Configure the Interface UnWedging Feature

The basic steps to set up an interface to be unwedged are as follows:

Step 1 Ensure that all the features described in this section are present in your image.

Step 2 Establish a system memory resource policy and apply it using the following commands:

resource policy

```
policy policy-name [global]
    system
```

Refer to the ERM Product Feature Guide, “Embedded Resource Manager” (URL listed in Table 20-5) for more information on the keywords and input values for the commands to set up a memory resource policy. You will apply the resource policy in Step 4, after setting up the Memory Resource Owner thresholds in policy node configuration mode in Step 3.

Step 3 Configure the Memory Resource Owner to trigger the Garbage Detector when interface memory use reaches a certain threshold, using the following command:

memory io

```
critical rising rising-value [interval rising-interval-value] [[falling] falling-value
    [interval falling-interval-value]]] [global]
```

The **global** keyword is not needed here if the **resource policy** is defined as **global**. Refer to the Product Feature Guide, “Interface Input Queue Memory Reclamation” (URL listed in Table 20-5) for more information on the keywords and input values for this command.

Step 4 Apply the system-wide resource policy set up in Step 2, in ERM configuration mode, to activate the global thresholding as follows:

user global policy-name

Step 5 Configure Flexible Packet Matching policy map filters on the wedged interface to identify and drop packets that match the filter constraints, using the commands shown below. Only **access-control** policy maps with the **drop** action are supported; other policy maps result in an “Incorrect policy-map” error. The packet-matching filter can use the **match field** command if a Protocol Header Description file is provided with the **load protocol** command.

```
load protocol location:filename
```

```

class-map type access-control class-map-name match-all
match field protocol protocol-field {your match criteria ...}
policy-map type access-control policy-map-name
class class-map-name
drop

```

Refer to the Product Feature Guide, “Flexible Packet Matching” (URL listed in Table 20-5) for details on the keywords and input values for these commands.

- Step 6** Configure the interface to reclaim the dropped packet buffers (assumed to have been leaked) that have been queued for longer than *secs* seconds using the following command, described in subsection 20.6.2.2, “memory debug leak(s) reclaim Command”:

```
memory debug leak(s) reclaim filter filter-name ... packet-age secs deallocate [exclusive]
```

The buffers marked as those to be reclaimed are disassociated from the interface’s input queue and released, allowing the input queue to be unwedged. It is considered safer to invoke the Garbage Detector to deallocate leaked packet buffers (omit the **exclusive** keyword) because it identifies valid leaks. However, the Garbage Detector is CPU-intensive, so the **exclusive** option allows the memory to be reclaimed with less impact on system operation.

The next subsection shows a configuration file example. The configuration parameter values are highly subjective and depend on the platform and site-specific usage trends, which should be carefully monitored before configuring this feature. For example, if a particular system’s usual IO memory consumption is only about 50%, and you observe a usage increase to 60% that you suspect might be due to a buffer leak, then you might want to configure the **rising** threshold at 60%. However, set the trigger thresholds carefully to avoid invoking the CPU-intensive Garbage Detector too often and adversely affecting system performance.

20.6.2.4 Interface Unwedging Configuration File Example

The following Cisco IOS configuration file segment shows the commands to configure the unwedging feature on the interface named Ethernet3/0. It defines the access control policy map P1 with the drop action for class C1 packets, and, at 90% memory in use by buffers, triggers the Garbage Detector to release dropped packets that are older than 180 seconds.

```

resource policy
  policy mem_policy global
    system
      memory io
        critical rising 90 interval 5 falling 10 interval 5
      !
      !
      !
    user global mem_policy
  !
  load protocol ip.phdf
  !
  class-map type access-control match-all C2
  match field IP tos eq 224
  class-map type access-control match-all C1
  match field IP tos eq 224
  !
  !

```

```
policy-map type access-control P1
    class C1
        drop
policy-map type access-control P2
    class C2
    log
!
!
interface Ethernet3/0
ip address 1.1.1.2 255.255.255.0
no ip route-cache
duplex half
memory debug leak reclaim filter P1 packet-age 180 deallocate
!
```

20.7 Links to Other Debugging Documentation

Following is a list of links to sundry documentation that might be helpful for debugging. Most of the links were provided by Paul Klepac in response to a plea on the `software-d` alias for links to useful debugging information. If you have any to add, send them to `ios-doc`.

- Using GCC
http://wwwin-swtools.cisco.com/swtools/cygnus/97r1/2_GNUPro_Compiler_Tools/Using_GNU_CC/gcc.html
- MIB-Police
http://wwwin-eng.cisco.com/Eng/VIOS/SNMP_WWW/mib-police.html
- Internetworking Terms and Acronyms
<http://www.cisco.com/univercd/cc/td/doc/cisintwk/ita/index.htm>
- ISDN Glossary
<http://www.cisco.com/warp/public/129/27.html>
- Multi Service Switching Business Unit
<http://wwwin-eng.cisco.com/Eng/WAN/WWW/operations/homepage/index.html>
- Voice Design Guide
http://wwwin.cisco.com/cmc/cc/so/cuso/epso/entdes/voice_dg.htm
- VoIP Implementation Guide
<http://wwwin-people.cisco.com/mcaranza/MyHome/VoIPManual/Guide/Index.htm#Configuring>
- IOS Memory Leaks

The Garbage Detector (GD) is a tool that can be used to detect memory leaks in IOS:

http://wwwin-eng.cisco.com/Eng/IOSTech/IOSInf/SW_Specs/Garbage_Detector_User.doc

The functional specification for the Garbage Detector feature is at:

http://wwwin-eng.cisco.com/Eng/SET/SES/GDinIOS_GEMII_sw_func_spec.doc

A comprehensive User's Guide for the Garbage Detector feature, called the Memory Leak Detector in technical support documentation, is available at:

http://www.cisco.com/univercd/cc/td/doc/product/software/ios124/124cg/hcf_c/ch45/hmleakd.htm

- IOS Memory Corruption
<http://wwwin-eng.cisco.com/Eng/IOSTech/WWW/MemCorrupt/>
- RSP, RP, and VIP2 Crashinfo Page
<http://wwwin-eng.cisco.com/Eng/Core/WWW/Presentations/LifeSavers/crashinfo.html>
- ISDN Tutorial
<http://www.ralphb.net/ISDN/>
- Chapter 10 of the common tools guide, “Debugging Software”:
<http://wwwin-enged.cisco.com/common/doc/tools/debugger.html>
- Student guide in an Engineering Education course on Cisco IOS Memory Troubleshooting:
<http://wwwin-enged.cisco.com/common/courses/mem/>
- Engineering Education course, “Debugging Cisco IOS Images”:
http://wwwin-enged.cisco.com/common/courses/cd_dii.htm
- IOS device drivers online documentation, *Cisco IOS Network Interface Drivers: Fundamentals of Architecture*:
http://wwwin-enged.cisco.com/doc/ios/nifd/12_3/
- Home URL for Engineering Education documentation and training:
<http://wwwin-enged.cisco.com/>
- URL for How to Ask Questions The Smart Way:
<http://www.catb.org/~esr/faqs/smarty-questions.html>

20.8 The Event Trace Facility

The event tracer subsystem is a general debugging subsystem which helps development engineers trace modules. It also provides support for identifying and storing the recent events and state changes more easily. The event tracer subsystem provides general hooks that can be put in the subsystem to be traced, which provides general tracing filters. The hooks then trace the events based on the filters supplied. The saved debug information can be extracted and then analyzed. This will help in keeping a log of the recent activities of the subsystem while maintaining a minimal overhead in doing so. The real advantage of event tracer is that it stores the data in binary format without applying any formatting or processing. This is especially good for tracing events that generate a lot of data very quickly, as well as enabling you to keep event tracer running during the router’s normal operation without affecting the router’s performance. The traced data can also be stored in a file. This file can later be moved out of the box and different formatting can be applied to it for further analysis.

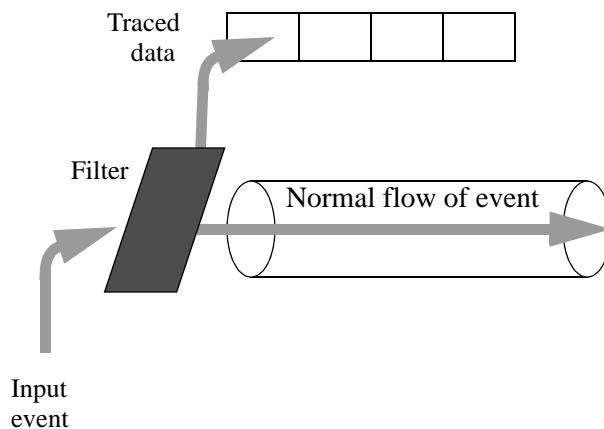
20.9 What is the Event Tracer?

Event tracer is a combination of a fast mechanism that logs a few bytes of debug information in a buffer area and a display mechanism that extracts and decodes this debug data. The overhead of doing the logging is minimal, so that it is feasible to have an event trace at least wherever one might have a conditional `buginf()`.

Event tracer is a cyclic buffer area with small, fixed format entries. Every time a message is received or transmitted, it is presented to the trace mechanism. A few quick checks are made to see whether the message should be traced, for example, is tracing enabled for this kind of message. If the message is to be traced, the first few bytes of the message are placed in the next position in the cyclic buffer, along with the timestamp, thereby overwriting the oldest entry.

Figure 20-1 shows the flow of event tracer, illustrating how the traced data is filtered off into the cyclic buffer area.

Figure 20-1 Flow of Event Tracer



To view the event trace, the show routine starts with extracting the oldest entry (or, if a start time is specified, the entry from that time). The timestamp and the few bytes of the message are decoded and printed. The show routine then extracts the next entry, prints that, and repeats until all entries have been shown.

The trace is not locked while it is being displayed; new traces can accumulate while old ones are being displayed. If trace entries get written more quickly than they are displayed, some entries will be lost and the show routine will indicate this and carry on. If there are excessive number of lost messages the show routine assumes that trace entries are being produced faster than it can consume them and for safety will give up trying.

To display the latest entries in a particular trace buffer every three seconds without entering the show command repeatedly, use the **monitor event-trace foo continuous [cancel]** EXEC command.

20.9.1 Event Trace Facility Key Features

The following list includes the key requirements/features for the basic mechanism of event tracer:

- Each data item stored in the tracer buffer has a time stamp stored which is recorded as close as possible to actual occurrence of the event.
- Each software component using the event trace assigns a unique component ID to its instance of event trace.
- Each data item stored in the trace buffer has a counter. This is a monotonically increasing value which is global across all the traces. This counter, along with a component ID, can be used for merging traces from different components and for event correlation.

What is the Event Tracer?

- Event tracer provides a baseline set of commands available on all trace events. Examples of this are “enable,” “disable,” “clear,” and “size.” The advantage of this is a consistent interface. The implementation of these commands in event tracer relieves the application programmer from maintaining buffers to store the debug information and devising effective ways to display and use this information.
- Event tracer also provides a “one-shot” mode, which clears the trace, sets it running, and then automatically disables it at the point where it would otherwise wrap and overwrite the buffer data.
- The baseline CLI is easily extensible on a per-trace basis. For example, GSR mbus trace could allow for tracing to be controlled by message type and by slot. This lets the application programmer customize and refine the tracing for that specific module. Users have to extend the CLI and also pass the filter to event tracer. Event tracer applies the user-supplied filter whenever it is invoked with the data to be traced.
- Through a CLI command, event tracer can write all the buffer data presently in the memory into a file. This allows for the easy copying of event data to an external machine for analysis.
- A software module can have multiple traces open. When a particular trace is initialized, the event tracer module returns a unique handle and this handle will be passed to APIs for storing the trace data.
- You can dump all the traces open in the system, either to the console or to the filesystem, through a single CLI command.
- You can enable or disable the storing of stack trace with trace buffer entries either with the CLI command or by specifying it during trace buffer initialization.
- The **show monitor event-trace merged all-traces** CLI command is available to show the entries in all trace buffers merged and sorted by timestamp.
- Using the **monitor event-trace foo continuous [cancel]** EXEC command, you can periodically display the latest entries in a particular trace buffer. This periodicity is currently set to three seconds.
- The format of timestamps displayed is user configurable. You can set it to display either the uptime or different date time formats.

20.9.2 Event Tracer Subsystem Files

The event tracer subsystem is implemented in the following files in the `/vob/ios/sys/os` directory, with the exception of `event_trace.h`, which is stored in the `/vob/ios/sys/h` directory:

- `event_trace.c`—This file contains the implementation of basic APIs exported by the event tracer subsystem. This module contains APIs to initialize the event tracer for a particular software subsystem, to write to a particular event trace buffer, to resize the buffer, to print the buffer and to free the event trace buffer.
- `event_trace.h`—This file is stored in the `/vob/ios/sys/h` directory and contains various data structures exported by the event tracer and also used by it. This includes a handle, which is associated with each instance of the event trace. This handle contains various parameters indicating the size of the trace and indices to write and display events. Data structure of an individual trace element is also in this file. When event trace is dumped into a file in filesystem, a header is written first indicating the component ID and individual record size. This data structure is also stored in this file.

This file contains the definition of the macro `EVENT_TRACE_BASIC_INSTANCE`. This macro expands into a chunk of code which handles all the basic CLI commands for the event trace. These CLI commands are shown in section 20.13 “Event Tracer CLI Commands.” This code also handles all the basic enabling, disabling, and printing of the event trace.

- `event_trace_private.h`—This file contains structures and definitions internal to event trace subsystem.
- `event_trace_chain.c`—This file contains routines to initialize the parser routines for event tracer. This puts in appropriate link points for event trace subsystem in the parser chain.
- `cfg_event_trace.h`—This file contains parser macros to set up event trace in config mode.
- `exec_event_trace.h`—This file contains parser macros to set up event trace in exec mode.
- `exec_show_event_trace.h`—This file contains parser macros to set up event trace in show mode.

20.9.3 Event Trace Facility Restrictions and Limitations

When a component is setting up the event trace, it has the following restrictions:

- Maximum name of event trace is 30 characters.
- Maximum size of an individual trace element is 256 bytes.
- Maximum number of trace elements in a particular instance of trace is 65535.
- Maximum size of filesystem URL to dump the current entries of the event trace in memory is 100 characters.

Before using the event trace a component has to set up a define for its component ID, and a few defines to extend the CLI if desired. Also a trace hook function and, optionally, a pretty format function need to be defined. Refer to section 20.10.1 for more details about how to set these up.

20.10 How to Use the Event Tracer

Subsystems which need to use the event tracer described in this document need to write filter functions. These functions will be put at various points in the code to be traced. If desired, subsystems using event trace can write extensions to basic CLI. Also, optionally, they can provide a pointer to a pretty format function. This function will be used to format the individual trace entry before displaying to the console.

Section 20.10.1 describes an example setup of event tracer for the IPC subsystem.

20.10.1 Example Setup of the Event Tracer

An application using the event trace needs to create two files, namely `xxx_trace.c` and `xxx_trace.h`. Suppose the IPC subsystem wants to use event tracing. Two files will be created for IPC trace:

`ipc_trace.c`—This file contains a filter function which is a hook placed at various places in the code where tracing is to take place. It may also contain extensions to the CLI and a pretty format function.

`ipc_trace.h`—This file contains the interfaces exported by this instance of trace and will be inserted at different points in the code to be traced. This header is included by any files where code should log to the new trace. This file typically contains the filter hook function.

20.10.1.1 Addition of Component ID

One of the initial steps in setting up the new event trace module is defining the component ID in `os/event_trace_ID_list.h`. The component ID is an arbitrary unique number that is used only by the event trace subsystem. For IPC this is defined as follows in `os/event_trace_ID.h`.

```
/*
 * Component IDs of components using trace
 */
enum {
    EVENT_TRACE_COMP_IPC,
};
```

20.10.1.2 Framework of ipc_trace.c

The framework of `ipc_trace.c` involves several steps:

20.10.1.2.1 Instantiating an Event Trace

The event tracer has a basic CLI. If you do not want to extend the CLI for the module that is using the event tracer, then define the `EVENT_TRACE_BASIC_INSTANCE` macro to create an instance of the event trace. The parameters for `EVENT_TRACE_BASIC_INSTANCE` are listed in Table 20-7.

Table 20-7 `EVENT_TRACE_BASIC_INSTANCE` Parameters

NAME	Prefix for the trace, typically “foo”
DESC	Description for parser help
COMP_ID	Component ID of the trace
ENTRY_SIZE	Size of individual trace entries
NUM_ENTRIES	Default number of entries in the trace
FLAGS	Different default conditions for the trace, such as trace on or store stacktrace
PRINT_FN_PTR	Pointer to pretty format function. Could pass NULL if no pretty formatting required

For example to create IPC trace, define the macro as shown here:

```
EVENT_TRACE_BASIC_INSTANCE (ipc, IPC event trace, EVENT_TRACE_COMP_IPC,
                           IPC_TRACE_ENTRY_SIZE,
                           IPC_TRACE_NUM_ENTRIES,
                           TRACE_ON|TRACE_STACKTRACE,
                           pretty_print_trace_entry)
```

This macro expands into a chunk of source code which handles all the interfacing with the event trace APIs and calls them appropriately. This also expands into code which handles all the basic CLI commands and sets up the parser chain for exec, show, and config modes.

20.10.1.2.2 Writing the Trace Hook Function

Next a function is written which is used as a hook and put at points where we want to trace data. An example of this is:

```
void xxx_trace_write (xxx_message *msg)
{
    char *elem;

    /*
     * Here the module can do finer tracing based upon different parameters.
     * For example examining the message, filtering can be done based upon
     * slot id, message id, etc. All these information about what parameters
     * to do finer tracing is maintained in this file and need to be written
     * by the developer using event trace for his/her module.
     *
     * Next get a slot from the trace buffer.
     * In xxx_trace_ctrl, replace xxx with NAME that was used in
     * EVENT_TRACE_BASIC_INSTANCE above, for example, ipc_trace_ctrl.
     */
    if (event_trace_get_slot(&xxx_trace_ctrl, &elem) != TRACE_NO_ERR)
        return;

    /*
     * copy whatever you want into elem
     */
    memcpy (elem, msg, XXX_TRACE_ENTRY_SIZE);
}
```

20.10.1.2.3 Writing the Pretty Format Function

Optionally a pointer to a pretty format function can be provided when declaring the macro EVENT_TRACE_BASIC_INSTANCE. This function will be used to pretty format the individual trace entries when displaying to the console or when dumping to the filesystem. If no such function pointer is provided that is, NULL is passed, then a hexdump of trace entries will be displayed to the console. The prototype of the pretty format function should be:

```
boolean pretty_format_func(char *p_input, char *p_output);
```

A return value of TRUE indicates that the entry needs to be printed; FALSE means don't print the entry.

Note p_output points to the pretty-formatted output buffer. The maximum size of the output buffer is 256, so the pretty format function should be careful not to overflow the buffer.

20.10.1.3 Framework of ipc_trace.h

This file contains the prototype of the function(s) exported by ipc_trace.c. For the simple case this would be:

```
void ipc_trace_write (ipc_message *msg);
```

20.10.1.4 Modification of the makefile

Amend the appropriate makefile to add the ipc_trace.c in the build.

20.11 Event Tracer API Functions

This section lists the event trace API functions, a description of their usage guidelines, and their algorithmic functionality. It also includes a short description of the initialization of event trace, and the initialization of subsystems using event trace.

20.11.1 Initialization

The event trace subsystem is initialized during the initialization of Kernel Class of subsystems. At this time event trace CLI is inserted in the parser chain and link points are added for event trace.

A subsystem which is using the event trace will during initialization insert its name and CLI extensions into the link points created above. It will also insert its control structure to the global list containing pointers to all the traces in the system.

For more information about CLI extensions, see section . For more information about control structures, see section 20.12 “Event Tracer Data Structures.”

20.11.2 Opening an Event Trace

The module being traced calls the `event_trace_open()` API function if that particular trace is being enabled for the first time. `event_trace_open()` creates a private trace buffer and returns a handle. This handle is used by that module for subsequent operations on this trace buffer.

In the case when trace is not enabled for the first time, a flag which maintains the state of the trace will be set indicating it to be enabled.

The following list describes the algorithmic functionality of `event_trace_open()`:

- Check the range of arguments passed in the API. Check for the length of the trace to be opened, and the size and the number of trace elements. In case of error, display an error message and return.
- Allocate memory for the handle associated with this particular trace.
- Calculate the size of trace elements so that it will be aligned in memory. Trace data is aligned with a multiple of ulonglong.
- Allocate the event trace data area. Memory allocated will be memory-aligned size of individual trace calculated above times the number of entries in the trace.
- Initialize the handle with trace name, number of entries, size of entries, component ID, and various indices for buffer manipulation.

20.11.3 Enabling in One-shot Mode

Event traces can also be enabled in one-shot mode. In this mode, the trace is cleared up, then events are recorded until the trace buffer reaches the wrap point and then disables further tracing. Tracing module calls the `event_trace_one_shot()` function to go into one-shot mode.

The following list describes the algorithmic functionality for the `event_trace_one_shot()` function:

- Clear the event trace.
- Set the variable in the handle `one_shot` to be TRUE.

20.11.4 Disabling an Event Trace

A particular instance of trace is disabled by turning off the flag which maintains the state of that trace. This flag is local to that particular module and no APIs of event trace are called.

20.11.5 Storing an Event

When an event needs to be stored, the module which is being traced calls the `event_trace_get_slot()` function. This returns a pointer to the data portion of a trace entry. The application can then store the event in whatever formats it wants. The application will have to make sure that it does not overstep its bounds and write event data which is greater than the size which it specified during trace initialization.

The following list describes the algorithmic functionality of the `event_trace_get_slot()` function:

- Verify whether the handle passed is correct or not.
- Disable interrupts.
- Get a pointer to a slot using the current index in the handle.
- Check whether the index has reached the maximum value. If yes, then,
 - Set the index to zero and increment the wrap count (count of how many times this trace buffer has reached its max value)
 - Check whether one-shot mode is enabled. If yes, then set the return status to be `TRACE_DISABLE` and clear the one-shot flag.
- Otherwise increment the index to get the next slot when called next time.
- Enable interrupts.
- Get current time and store it in the header of slot entry.
- Get a counter value and store it in the header of slot entry. This counter is a monotonically incrementing value which is global to all traces. This counter is used for event correlation between different traces.
- Return a pointer to data portion of the slot entry back to be filled by the module calling trace.

20.11.6 Displaying Event Trace Entries

Event trace entries in a buffer can be displayed onto the console with the `event_trace_print()` and `event_trace_print_all()` functions. The module being traced calls `event_trace_print()` with a pointer to a function as one of the arguments which will apply formatting to the individual trace entries and display it to the console. If no formatting function is provided, then trace is hex dumped to the console.

The following list describes the algorithmic functionality of the `event_trace_print()` function:

- Verify whether the handle passed is correct or not.
- If CLI command was to print the latest entries in the trace, then
 - Get an index and wrap count of the oldest entry in the trace.

- If (*wrap count* < *last_disp_wrapcount*) then index and wrap count of the element to be displayed is that which is stored in variables *last_disp_index* and *last_disp_wrapcount* in the handle. Else if (*wrap count* == *last_disp_wrapcount*) index of the element to be displayed is equal to the greater of *last_disp_index* and the index of oldest entry in trace. Else we will have to start displaying from the oldest entry in the trace.
- Else,
 - Calculate start time for the trace elements to be displayed
 - Get an index and wrap count of the oldest entry in the trace.
- For (; ;)
 - Get an element specified by index and wrap count. If that entry is lost it returns the next oldest entry in the trace. Index and wrap count is incremented to point to next entry.
 - Switch (return value)
 - If (EVENT_TRACE_LOST), print out a message that one or more entries are lost and increment the count for lost entries. Then follow through to next step.
 - If (EVENT_TRACE_NORMAL), print the timestamp and counter of the entry. Then check whether a pretty formatter function was passed as an argument. If yes, then call that function to pretty format the entry, and then print it. Otherwise, hexdump the trace entry.
 - If (EVENT_TRACE_FINISHED), we are done with print, return back to the caller.
 - If lost entries is greater than 20 and stdio->tty_length == 0 then bail out by printing tracing being produced too quickly to be printed.
- End of for loop

20.11.6.1 Printing All the Event Traces

The following list describes the algorithmic functionality of the `event_trace_print_all()` function:

- Get pointer to the head of list containing control structure pointers to all the traces in the system.
- While (control structure pointer != NULL)
 - Get the pointer to the handle of the trace
 - Call the API to print the particular instance of event trace pointed to by the handle. (The algorithm is do so has been explained above)
 - Get the next pointer in the control structure list.
- End while

20.11.7 Changing the Event Trace Buffer Size

The module being traced, when it initializes the trace, specifies the number of entries in the event trace. The tracing module can change this during run-time by calling the `event_trace_resize()` function. If the number of entries is decreased or increased, a new trace space is allocated and old events are copied into the new trace buffer.

An event trace can be resized by increasing or decreasing the number of entries. The arguments passed to the API to resize the event trace are pointer to the pointer to old handle and the new size of event trace.

The following list describes the algorithmic functionality for the `event_trace_resize()` function:

- Open a new trace buffer. Name and Component ID of this new trace is same as the old trace. Number of entries is whatever specified by the user.
- If the return value from the above operation contains an error, this probably means that we don't have enough memory to resize the trace. So just return. We will continue to have the old trace before the resize.
- Else, if we have successfully created a new trace area we will have handle to the new trace in `new_trace_handle`
 - `loopcount=0`
 - Retry:
 - Set index and wrap count of `new_trace_handle` to be zero.
 - Now copy entries from old trace to new one, one at a time:
 - Get an index and wrap count of the oldest entry in the old trace.
 - For (;;)
 - Get pointer to slot entry from the new trace.
 - Get an element specified by index and wrap count from the old trace and copy it into the new trace slot. The return value of the get data from the old trace is stored in the variable `elem_result`.
 - If (`elem_result == EVENT_TRACE_LOST`)
 - Goto Retry
 - If (`elem_result == EVENT_TRACE_FINISHED`)
 - break
 - Check whether the index has reached the maximum value for `new_trace_handle`. If yes, then,
 - Set the index to zero and increment the wrap count (count of how many times this trace buffer has reached its max value) for `new_trace_handle`.
 - Otherwise, increment the index of `new_trace_handle`.

We might be copying a lot of data—be nice to others:

- `if (++loopcount > 1000)`
- `loopcount = 0;`
- `process_suspend();`
- Close the old trace and release all the memory associated.
- Return the pointer to the new trace handle.

20.11.8 Clearing the Event Trace

Module being traced calls the `event_trace_clear()` function to clear the entries in the event trace.

The following list describes the algorithmic functionality for the `event_trace_clear()` function:

- Disable interrupts.

- Set index and wrapcount of the trace to be equal to zero.
- Enable interrupts.
- Also set last displayed index and wrapcount variables in the handle to be also equal to zero.

20.11.9 Storing the Event Trace in the Filesystem

Current events traces stored in memory can be saved in the filesystem. Software modules call `event_trace_dump()` with URL of the location where to store the trace dump.

The following list describes the algorithmic functionality for the `event_trace_dump()` function:

- Verify whether the handle passed is correct or not. Also verify whether the URL of the filesystem is passed to where the event trace will be dumped.
- Open the file in the filesystem.
- Write the file header into the file. The header contains Component ID and record size of each individual trace element.
- Get an index and wrap count of the oldest entry in the trace.
- For (; ;)
 - Get an element specified by index and wrap count. If that entry is lost it returns the next oldest entry in the trace. Index and wrap count is incremented to point to next entry.
 - Switch (return value)
 - If (`EVENT_TRACE_LOST`), print out a message that one or more entries are lost and increment the count for lost entries. Then follow through to next step.
 - If (`EVENT_TRACE_NORMAL`), then check whether pretty formatter function pointer was passed. If yes, then call the pretty formatter function and then write the output to the file. Otherwise, write the trace entry in the binary form into the file.
 - If (`EVENT_TRACE_FINISHED`), we have finished writing to the file; return back to the caller.
- End of for loop

20.11.9.1 Dumping All the Event Traces into the Filesystem

The following list describes the algorithmic functionality for the `event_trace_dump_all()` function:

- Get pointer to the head of list containing control structure pointers to all the traces in the system.
- While (control structure pointer != NULL)
 - Get the pointer to the handle of the trace
 - Call the API to dump the particular instance of event trace pointed to by the handle. (The algorithm is do so has been explained above)
 - Get the next pointer in the control structure list.
- End while

20.11.10 Closing an Event Trace

The following list describes the algorithmic functionality for the `event_trace_close()` API function:

- Verify whether the handle passed is correct or not.
- Disable interrupts.
- Copy the pointer to the handle to a local variable '`temp_handle`'
- Set the pointer to handle to be `NULL` so as to disable it.
- Enable interrupts
- Now the local variable `temp_handle` contains the handle of the closed trace. Free up the trace data area and then the memory occupied by the handle.

20.11.11 Inserting a Control Structure of an Event Trace into a Global List

The following bullets list the algorithmic functionality for the `event_trace_insert_node()` function:

For more information about control structures and global lists, see section 20.12 “Event Tracer Data Structures.”

- If (pointer to global list == `NULL`)
 - Allocate memory for a node in control structure list
 - Point the data in the node to the pointer to control structure passed.
 - Set the next pointer of the inserted node to be `NULL`
- Else,
 - Loop the list to get to the end of the list
 - Allocate memory for a node in control structure list
 - Point the data in the node to the pointer to control structure passed.
 - Insert the node at the end of the list
 - Set the next pointer of the inserted node to be `NULL`

20.12 Event Tracer Data Structures

Each instance of an event trace is controlled by a control structure. This structure has a string containing the name of the trace as well as an unsigned integer containing the component ID. Each subsystem using event trace has a unique component ID. Component ID is used for event correlation between different sets of event traces from different components. Event traces can be stored off line and then they can be merged based on component ID as key. Other variables indicate the status and various states of that particular instance of event trace.

```
/*
 * Control structure for individual trace instances
 */
typedef struct event_trace_control {
    char trace_name[EVENT_TRACE_MAX_NAME]; /* trace buffer name */
    uint comp_id;                         /* Component ID of trace */
```

```

boolean trace_enabled;           /* current state of trace */
boolean trace_default;          /* default state of trace */
void    *trace_handle;          /* pointer to the trace handle */

boolean trace_on_off_cfgd;      /* whether config thru config mode */
boolean trace_size_cfgd;        /* whether size has been configured */
uint   trace_configured_size;   /* new configured size */
boolean trace_dump_file_cfgd;   /* whether dump filename has been cfgd */
char   trace_dump_file[EVENT_TRACE_MAX_FILENAME];
uint   trace_on_off_flags;      /* Reasons the trace may be on or off */

pretty_format_func pretty_format_fn_ptr; /* ptr to pretty format function */
} event_trace_control_t;

```

When an instance of event trace is created during initialization, a pointer to its control structure is put into a global list. This list contains pointers to all the event traces in the system. This is used when all the event traces in the system need to be displayed or dumped to the filesystem.

```

/*
 * Individual nodes of list containing pointers to all event tables created
 */
typedef struct event_trace_global_list {
    event_trace_control_t *p_control;
    struct event_trace_global_list *next;
} event_trace_global_list_t;

```

The event trace is cyclic in nature. It is controlled by a structure containing a pointer to the trace area, an index that indicates where in this area the next entry is to be made, a size to indicate when tracing wraps back to the start, and a count of the number of times a wrap has been made.

```

/*
 * Handle
 */
typedef struct event_trace_handle {
    boolean one_shot;           /* TRUE if one_shot enabled */

    uint index;
    uint max_index;
    uint sizeof_elem;           /* size of elem as specified in trace_open() */
    uint sizeof_elem_align;     /* size of elem after alignment */
    uint wrapcount;

    uint last_disp_index;       /* index of last display entry */
    uint last_disp_wrapcount;   /* wrapcount of last display entry */

    char *trace_data;           /* pointer to event trace data */
} event_trace_handle_t;

```

The Boolean `one_shot` indicates whether the trace is in one-shot mode or not. If in one-shot mode, the event trace buffer is filled until the wrap point and then disabled.

20.12.1 Examining Event Trace Data

The wrap count is used when examining the trace entries. Say we start by printing the oldest entry. We remember the wrap count and index that defines the next entry. We then proceed to the next entry and print that. While this is happening, additional entries may be made to the trace and it is possible that new traces are produced faster than old ones are consumed in printing. In this case, it is possible

that the entry that was next to be printed is reused for a new trace. If this happens, the output routine can detect the missed message(s), indicate some messages have been dropped, and move on to output details of whatever trace is currently the oldest.

In summary, a trace element is defined by a particular value of the wrap count and index. When printing a trace entry, the wrap count and index for the next entry in sequence are remembered and used later to check that the trace element is still valid.

Variables `last_disp_index` and `last_disp_wrapcount` store the index and wrap count of the last event displayed to the console. These are used when the user is only interested in looking at the newly generated events since the last display.

Each event trace data is prefixed with a header which contains timestamp for the entry and a monotonically incrementing counter. This counter is global across all the traces in the system. This can be used for event correlation of different traces.

```
/*
 * The trace element prefix for each individual trace.
 */
typedef struct event_trace_elem_prefix {
    sys_timestamp timestamp;
    ulonglong     counter;           /* monotonically incrementing count
                                         for event correlation */
    uchar data[0];                  /* event data */
} event_trace_elem_prefix_t;
```

When current elements in the event trace are stored in the filesystem, the stored file is prefixed with a header. The header contains the component ID of the subsystem and the size of individual trace including the header.

```
/*
 * Header of the trace file written to the filesystem
 */
typedef struct event_trace_file_header {
    uint comp_id;
    uint record_size;             /* this is sizeof_elem + sizeof prefix */
} event_trace_file_header_t;
```

20.13 Event Tracer CLI Commands

Event tracer provides a basic set of CLI commands to enable, disable, configure the size of, and print the event trace. Individual modules can extend the CLI for their modules. The basic CLI commands are described briefly in this section.

20.13.1 Exec Mode Commands

The following list includes the exec mode commands for the trace:

```
gt10-12k-2#monitor event-trace ?
  all-traces      dump all the event traces
  gsrha_all       GSR HA generic event trace
  gsrha_driver    GSR HA driver event trace
  gsrha_oir       GSR HA OIR event trace
  gsrha_swover   GSR HA switchover event trace
  gsrha_sync      GSR HA sync event trace
  lblsync         LBL Config Sync event trace

gt10-12k-2#monitor event-trace gsrha_driver ?
  clear           Clear the trace
  continuous     Continuously display latest event trace entries
  disable         Disable tracing
  dump            dumps the event buffer into a file
  enable          Enable tracing
  one-shot        Clear trace, set running, then disable at wrap point

gt10-12k-2#monitor event-trace all-traces ?
  continuous     Continuously display latest merged event trace entries
  dump           dump all the event traces
```

The following command displays the latest entries in the gsrha_driver trace buffer every three seconds:

```
gt10-12k-2#monitor event-trace gsrha_driver continuous
```

To disable continuous display, enter the following command:

```
gt10-12k-2#monitor event-trace gsrha_driver continuous cancel
```

20.13.2 Exec Show Mode Commands

The following list includes some of the exec show mode commands for the trace related to displaying the buffer:

```
gt10-12k-2#show monitor event-trace gsrha_driver ?
  all           Show all the traces in current buffer
  back          Show trace from this far back in the past
  clock         Show trace from a specific clock time/date
  from-boot     Show trace from this many seconds after booting
  latest        Show latest trace events since last display
  parameters   Parameters of the trace
```

To display the current entries for the gsrha_driver trace buffer, enter the following command:

```
gt10-12k-2#show monitor event-trace gsrha_driver all
```

To display the trace entries in all trace buffers merged and sorted by timestamp, enter the following command:

```
gt10-12k-2#show monitor event-trace merged all-traces
```

To display some of the parameters specific to a trace buffer, enter the following command:

```
gt10-12k-2#show monitor event-trace gsrha_driver parameters ?
  size          Size of trace
```

20.13.3 Config Mode Commands

Some of the global config mode commands are:

```
gt10-12k-2(config)#monitor event-trace ?
  all-traces      configure merged event traces
  gsrha_all       GSR HA generic event trace
  gsrha_driver    GSR HA driver event trace
  gsrha_oir       GSR HA OIR event trace
  gsrha_swover   GSR HA switchover event trace
  gsrha_sync      GSR HA sync event trace
  lblsync         LBL Config Sync event trace
  sequence-number Display event trace entries with sequence number
  stacktrace      Display stack trace stored with event trace entries
  timestamps      Format of event trace timestamps
```

For example, the following shows the different options for setting the display format of timestamps:

```
gt10-12k-2(config)#monitor event-trace timestamps ?
  datetime      Timestamp with date and time
  uptime        Timestamp with system uptime
  <cr>
```

There can be various config mode commands depending upon the component, for example:

```
gt10-12k-2(config)#monitor event-trace gsrha_driver ?
  disable        Disable tracing
  dump-file      Set name of trace dump file
  enable         Enable tracing
  size           Set size of trace
  stacktrace     Trace call stack at tracepoints; clear the trace buffer first
```

For example, to set up the size for the trace buffer, enter the following command:

```
gt10-12k-2(config)#monitor event-trace gsrha_driver size ?
  <1-65536>  Number of entries in trace
```

20.14 What is the Enhanced Error Message Log Count

The logging facility on Cisco IOS allows the user to save important messages either locally or to a remote host. When the important messages exceed the capacity of the local buffer dedicated to storing the messages, the oldest messages are removed. The problem is that the customer may miss important messages due to failure to periodically check the logging buffer. The logging buffer may not be on at all, causing the customer to lose all information about the errors that may have occurred. In order to provide a Cisco IOS console user more information about messages that have occurred in the past and may have been removed from the local buffer, a counter will be created to count the occurrences of each error message and timestamp the last occurrence. Furthermore, these messages will be sorted by the message facility. Messages from each message facility will be grouped together and totaled in the count.

What is the Enhanced Error Message Log Count

20.14.1 End User Interface

20.14.1.1 How to Enable this Feature

To enable the enhanced error message log count, first enter configuration mode:

```
saa-ts 1-5#conf t
```

You will see the following sentences:

Enter configuration commands, one per line. End with CNTL/Z

Then type

```
saa-ts 1-5(config)#logging count
```

Then exit configuration mode:

```
saa-ts 1-5#end
```

20.14.1.2 How to Disable this Feature

To disable the enhanced error message log count, first enter configuration mode:

```
saa-ts 1-5#conf t
```

You will see the following sentences:

Enter configuration commands, one per line. End with CNTL/Z

Then type

```
saa-ts 1-5(config)#no logging count
```

Then exit configuration mode:

```
saa-ts 1-5#end
```

20.14.1.3 How to Verify Whether You Have this Feature or Not

To check whether you have this feature enabled or disabled, type

```
saa-ts 1-5#show logging
```

The output might be:

```
Syslog logging: enabled (0 messages dropped, 27 messages rate-limited, 0 flushes, 0 overruns)
Console logging: level debugging, 70 messages logged
Monitor logging: level debugging, 0 messages logged
Buffer logging: level debugging, 97 messages logged
Logging Exception size (8192 bytes)
Count and timestamp logging messages: enabled
Trap logging: level informational, 101 message lines logged
--More--
...
```

Notice the “Count and timestamp logging messages: enabled” line. This means that the enhanced logging feature is enabled.

The line would be “Count and timestamp logging messages: disabled” if the enhanced logging feature were disabled.

20.14.1.4 Using this Feature

Once you have enabled this feature, then you can use it to see the number of times a particular error message occurs and the timestamp of the last occurrence.

This feature inherits the time format for the show commands, based on the “service timestamps” setting.

Type the command

```
saa-ts 1-5#show logging count
```

The output then might be:

Facility	Message Name	Sev	Occur	Last	Time
SYS	CPUHOG	3	1	*Mar 23	21:03:28
SYS	NOMEMORY	1	10	*Mar 23	21:09:35
SYS	GETBUFFFAIL	1	5	*Jan 11	18:39:40
SYS	CONFIG_I	5	10	*Feb 19	11:02:34
SYS TOTAL			26		
LINK	CHANGED	5	20	*Jan 5	17:33:09
LINK	UPDOWN	3	9	*Feb 21	18:21:56
LINK TOTAL			29		
LINEPROTO	UPDOWN	5	55	*Mar 11	13:05:44
LINEPROTO TOTAL			55		

Then the user can drill down into the syslog for details without sifting through it all using “| include”.

For example, if the user is only interested in CPUHOG:

```
saa-ts 1-5#show log count | include CPUHOG
```

The output would be:

SYS	CPUHOG	3	1
-----	--------	---	---

Also, the user could see totals using the command:

```
saa-ts 1-5#show log count | include TOTAL
```

SYS TOTAL	26
LINK TOTAL	29
LINEPROTO	55

20.15 The Receive Latency Trace Facility

While Event Tracing captures a series of events with context data and time stamps, receive latency tracing captures the times between two trace points and is not dependent on context to be meaningful. For example, latency tracing is not concerned so much about whether an interrupt occurred, but rather, given that it happened, how much CPU time did it take to run.

20.15.1 Overview

The purpose of the Receive Trace Facility is to capture the time that is spent processing different types of packets during a receive interrupt. The intent is to identify which packet types may be impacting delay sensitive and mission critical traffic because they are taking too much time to process. This is accomplished by storing small amounts of packet information during a receive interrupt. The information is written into a circular buffer of data structures which for efficiency is stored as “raw” data, in the sense that no attempt is made to interpret or format the information.

There is a separate data buffer for each packet type and each buffer is dereferenced and processed one at a time by a periodic function that runs at process level. The periodic function has two tasks: to manage the array of data buffers, and to look for packet processing times exceeding a configured threshold, printing an error message if any are found.

The facility supports three user commands, one to configure the receive trace defaults, one to show the formatted trace data, and one to run tests on the media instance server and client API. No further user interaction is necessary since the periodic function manages the memory requirements and displays real time messages at a low output rate.

The Receive Trace Facility is designed to be a compile time feature that is built into a “special” image used for monitoring receive interrupt latency in time critical networks. It is not meant to be released in a production image because of the impact on receive interrupt performance. Thus, this facility is coded behind a compiler switch and has no dependencies outside the scope of the switch.

The “special” image is meant to be used in the field by customer engineers to monitor receive interrupt latency in time critical networks. For instance, if there is a Cisco customer with a network that has a lot of VoIP traffic and there is a lot of troublesome variation in the packet forwarding rates, the CE could load the “special” image to get a profile of the processing times for the various received packets and discover that several rx types are exceeding the default threshold for that particular platform, but that only a few are taking a really long time. The CE could then recommend that if just those few are dealt with, say by changing the configuration or by not allowing those types on that platform, the problem would be solved.

Finally, it is important to note that this facility is not designed for device testing and does not provide meaningful results under stressful situations. A platform running at high CPU usage could not give enough time to the periodic process to avoid data lossage, and the snapshots would show a very small slice of the total picture.

20.15.2 Storing Packet Information

There is no single common place to capture the processing time spent on every packet forwarded to Cisco IOS. A single interrupt may handle many packets collected in a hardware receive buffer and may service more than one interface type. There are many interface hardware types per media type and there are many combinations of media types per platform, and they all forward packets through specialized functions. In fact, the WAN media may specify a forward function that is dependent on the current configuration.

However, each platform has its own versions of “media” handlers which have all the necessary information to trace the time spent processing packets over a specific media. A handler determines the packet’s receive type and then decides how to forward the packet. It makes the call to fastswitch by protocol if possible or to coalesce particles (create a contiguous packet from a particle-based one by copying it into another piece of memory), if necessary, and to process switch if appropriate.

To trace a packet's processing time the "start" clock value is stored right after the handler determines the rx type. The "stop" clock value is then stored right after whichever forwarding call is made to the Cisco IOS from the media handler. It is desirable to know whether the packet was process switched or fastswitched, so the stop time is written into one of two fields, depending on the forwarding method.

All the tracing code for a particular media is declared within the scope of the file containing the platform media interrupt handler. The tracing code includes a global array of circular buffers with one `malloc`'d buffer per packet type. Each buffer has a header that includes the buffer size, a total count, an index mask, and a pointer to a list of entry structures.

The header structure is defined as follows:

```
typedef struct rx_trace_buffer_ {
    ulong          size;           // in bytes, including the header.
    ulong          count;
    ulong          mask;
    rx_trace_entry_st  entry[0];
} rx_trace_buffer_st;
```

The header is followed by a contiguous block of memory containing the entry structures. The entry list is declared as a zero length array and an entry is dereferenced as follows:

```
entry = &buffer->entry[index];
```

This scheme allows a new buffer of a different size to be dynamically allocated, since the buffer size is not hard coded.

The entry structure is defined as follows:

```
typedef struct rx_trace_entry_ {
    ulong  start_time;
    ulong  stop1_time;
    ulong  stop2_time;
} rx_trace_entry_st;
```

There are three inline functions that store the data:

```
entry = rx_trace_start(buffer_array[rxtype]);
rx_trace_stop1(entry);
rx_trace_stop2(entry);
```

These functions write into the correct trace entry. `rx_trace_start()` grabs the index, increments the counter, writes the start time, and returns a pointer to the entry:

```
static inline rx_trace_entry_st *rx_trace_start (rx_trace_buffer_st *buffer)
{
    long          index  = buffer->count++ & buffer->mask;
    rx_trace_entry_st *entry  = &buffer->entry[index];

    USEC_GET_TIMESTAMP(&entry->start_time);
    return (entry);
}
```

The other two functions then use the entry pointer to write the stop time. `rx_trace_stop1()` stores the stop time for process switched packets, and `rx_trace_stop2()` is the stop time for fastswitched packets:

```
static inline void rx_trace_stop1 (rx_trace_entry_st *entry)
{
    USEC_GET_TIMESTAMP(&entry->stop1_time);
```

The Receive Latency Trace Facility

```

    }

    static inline void rx_trace_stop2 (rx_trace_entry_st *entry)
{
    USEC_GET_TIMESTAMP(&entry->stop2_time);
}

```

On some platforms packets may need to be coalesced from particles into one contiguous packet, and sometimes this is done by the platform hardware. The receive handler gives the packet to a DMA copy engine and moves on. The DMA generates its own interrupt when it's done and calls a "post DMA" handler to forward the packet.

This post handler must be coded with the `stop1()` and `stop2()` tracepoints, and to support this the trace entry is passed from the receive interrupt handler to the DMA interrupt handler by way of a first-in first-out asynchronous entry buffer, called "efifo". The efifo buffer is implemented as a small circular buffer the size of a power of two, and an incrementing counter which is masked to find the index.

Here is the entry FIFO structure and some examples of the "coalesce" and "post" tracepoints that push and pop the FIFO:

```

typedef struct rx_trace_efifo_ {
    rx_trace_entry_st *entry[RX_TRACE_FIFOSIZE];
    ulong             head_ndx;
    ulong             tail_ndx;
} rx_trace_efifo_st;

static inline void rx_trace_coalesce (rx_trace_efifo_st *efifo,
                                     rx_trace_entry_st *entry)
{
    efifo->entry[efifo->head_ndx++ & RX_TRACE_FIFOMASK] = entry;
}

static inline rx_trace_entry_st *rx_trace_post (rx_trace_efifo_st *efifo)
{
    return (efifo->entry[efifo->tail_ndx++ & RX_TRACE_FIFOMASK]);
}

```

The tracepoints and buffers are then hidden behind macros so that the lines of code are removed by the compiler if the rx trace facility is not defined.

There is one "start" macro, two "stop" macros and two "coalesce" macros. Note that the letters "media" from below are meant to be generic, and that the actual macros and data structures would have the real media prefix, such as `amdfc`, `amdp2`, or `serial` (this also applies to the "MEDIA" macros with capital letters):

```

#ifndef RX_TRACE_FACILITY

extern rx_trace_buffer_st **amdfc_rx_trace_array;
extern rx_trace_entry_st   *amdfc_rx_trace_entry;
extern rx_trace_efifo_st   amdfc_rx_trace_efifo;

#define MEDIA_RX_TRACE_START(rxtype) \
    media_rx_trace_entry = rx_trace_start(media_rx_trace_array[rxtype])

#define MEDIA_RX_TRACE_STOP_PS  rx_trace_stop1(media_rx_trace_entry)

#define MEDIA_RX_TRACE_STOP_FS  rx_trace_stop2(media_rx_trace_entry)

```

```

#define MEDIA_RX_TRACE_COALESCE \
    rx_trace_coalesce(&media_rx_trace_efifo, media_rx_trace_entry)

#define MEDIA_RX_TRACE_POST_PS \
    rx_trace_stop1(rx_trace_post(&media_rx_trace_efifo))

#define MEDIA_RX_TRACE_POST_FS \
    rx_trace_stop2(rx_trace_post(&media_rx_trace_efifo))

#else
#define MEDIA_RX_TRACE_START(rxtype)
#define MEDIA_RX_TRACE_STOP_PS
#define MEDIA_RX_TRACE_STOP_FS
#define MEDIA_RX_TRACE_COAL_PS
#define MEDIA_RX_TRACE_COAL_FS
#define MEDIA_RX_TRACE_POST_PS
#define MEDIA_RX_TRACE_POST_FS
#endif

```

20.15.3 The Periodic Function

Each platform's media handler has its own periodic function which manages the trace buffers and reports any packets that exceed the configured maximum processing time. It begins executing at runtime and uses a "passive" timer to sleep periodically.

At runtime the periodic function initializes the `rx_buffer_array[]` with allocated trace buffers. The length of each buffer, which is the byte count of the header and all the entries, is determined by either the hard coded default or by the configured default, or by the periodic function itself. The hard coded default buffer size is meant to be platform dependent and consists of two values that represent a range of lengths from '`rx_min_entries + header_size`' to '`rx_max_entries + headersize`'. The number of entries is calculated to be a power of two, and the index mask is set to the number of entries minus one.

For example, to initialize the buffer size to the minimum number of entries:

```

buffer->mask = RX_TRACE_ENTRY_MIN - 1;

buffer->size = (RX_TRACE_ENTRY_MIN * RX_TRACE_ENTRY_SIZE)
               + RX_TRACE_HEADER_SIZE;

```

To increase the number of entries:

```

if (buffer->mask < RX_TRACE_ENTRY_MAX - 1) {
    buffer->mask = buffer->mask << 1 + 1;

    buffer->size = ((buffer->mask + 1) * RX_TRACE_ENTRY_SIZE)
                  + RX_TRACE_HEADER_SIZE;
}

```

This means that the number of entries can only be doubled (or halved) at any one time. The reason is for efficiency. The count field is an incrementing 'long' value that keeps the total number of received packets of an rx type. The interrupt handler calculates the offset to the next entry in the circular buffer by simply 'AND'ing the count field with the mask field and then incrementing the count:

```
index = buffer->count++ & buffer->mask;
```

The Receive Latency Trace Facility

This is a special case of “modulo”, where masking is very efficient but requires that the number of entries be a power of two and the mask be the number of entries minus one.

Once a buffer is allocated and initialized the periodic function monitors the index and calculates the wrap count in order to adjust the buffer size and the sleep period in an attempt to minimize data lossage from overwrites as the buffer cycles.

The combination of the number of entries and the sleep period determines the rate that buffer entries can be processed. The exact rate cannot be determined because of system scheduler latency and the interaction between the rx type buffers in the global array, but assuming that the minimum number of entries is thirty two and the default sleep time is one second, then the approximate maximum trace throughput is 32 pps.

In this scenario the periodic process wakes up every second and processes the entries captured by each rx type buffer. Lossage can be detected by subtracting the previous total count from the current total count and dividing by the number of entries. The action taken to manage lossage is as follows for each rx type buffer:

- if the buffer cycled once or more in one minute
 - double the buffer size
- if the buffer cycled twice or more
 - set minimum sleep time for this buffer
- else if the buffer is less than half full for one minute
 - increase the sleep time for this buffer
- if the sleep time equals the max default
 - half the buffer size
- set the actual sleep time to the lowest of all the buffers

The object here is to minimize the amount of time scheduled for, and the total memory allocated to the periodic process. This requires defining the minimum and maximum sleep times and buffer sizes. This, in turn, determines the range of memory allocated for the media type.

The relevance of time scheduled for the periodic process is in the trace buffer throughput; a short sleep interval means more scheduled time and more buffers processed. Since the highest packet rate is really a function of the media, the minimum sleep time has been arbitrarily chosen to be one tenth of a second. Similarly, the smallest buffer size should have enough entries to calculate meaningful snapshot values, so the minimum number of entries is arbitrarily set to be thirty two (a power of two).

From here the maximum number of entries can be calculated. Assuming a top forwarding rate of 20,000 packets per second at the minimum sleep interval of a tenth of a second, then the maximum number of entries is 2,000, or 2048 rounded up to a power of two. So the minimum and maximum allocated memory in bytes is as follows:

```
#define RX_TRACE_ENTRY_MIN    32
#define RX_TRACE_ENTRY_MAX    2048
#define RX_TRACE_ENTRY_SIZE   sizeof(rx_trace_entry_st)
#define RX_TRACE_HEADER_SIZE  sizeof(rx_trace_buffer_st)

min_mem = (RX_TRACE_ENTRY_MIN * RX_TRACE_ENTRY_SIZE)
          + RX_TRACE_HEADER_SIZE;

max_mem = (RX_TRACE_ENTRY_MAX * RX_TRACE_ENTRY_SIZE)
          + RX_TRACE_HEADER_SIZE;

minimum total memory =      524 bytes (0.8K aprox)
```

```
maximum total memory = 32,780 bytes (32K aprox)
```

This calculation assumes a single rx type, whereas on a real network there may be many types of packets. In fact there are sixty six defined rx types and if the packet rate is spread evenly among them and then rounded up to the nearest power of two, then the worst case scenario would be:

```
entries_per_type = (20,000 / 66) * .1
                  = 30.3 = 32 (rounded)

max_size = (entries_per_type * RX_TRACE_ENTRY_SIZE)
           + RX_TRACE_HEADER_SIZE;

max_mem = max_size * max_rx_types = 524 * 66;

worst case total memory = 34,584 = 33K
```

The selected packet rate of 20,000 pps is low for the high end platforms. Extrapolating up to an arbitrary forwarding rate of 80,000 pps gives a worst case memory usage of approximately 132K.

20.15.4 Time Threshold Messages

The periodic process cycles through the global rx buffer array, dereferencing each rx type buffer. It loops through the stored entries and subtracts the stored ‘start’ time from the ‘stop’ time values to get the time spent processing each packet. It then compares the results to the configured threshold and counts the number of packets exceeding the threshold, storing the longest processing time. After processing the buffer, if the threshold was exceeded then an error message is generated and printed to the console. The periodic process prints only one message per rx type per second. The reason for this is to prevent excessive output to the console.

To keep track of the error count and total count, as well as the longest time, the sleep time and the print rate, the periodic process creates a “status” buffer for each rx type that it encounters. These buffers are stored in a global `status_buffer_array[]` which is indexed by rx type. Note that a status buffer is not allocated for rx types that are never received.

Here is the definition of the status buffer:

```
typedef rx_trace_status_ {
    sys_timestamp period;
    sys_timestamp savedtime;
    sys_timestamp printtime;
    long          total;
    long          errors;
} rx_trace_status_st;
```

The following is a formatted error message example for ‘DODIP’ rx_type:

```
rx threshold error - 'AMD Ethernet' 'DODIP', proc-sw, time: 389, errors: 6,
total: 63 (lossage 0)
```

The message is meant to be a real time error indicator, though it is logged if message logging is enabled.

20.15.5 Snapshot Display

The snapshot display is a parser driven “show” function that works similarly to the periodic process. It cycles through the rx buffer array, dereferencing each rx type buffer and looping through the entries subtracting the stored start time from the stop time values to calculate the time spent processing each packet. The errors are counted and the longest time is saved but no error messages are generated. A summary is then printed for the trace buffer.

The result is a list of snapshot summaries printed for each `rxtypes` buffer, using the following format:

```
RX Trace Snapshot of circular buffers

SNAP_UNK proc-sw 00018, errors 00000, usec: min 00189, avg 00214, max 00237
...
...
```

Though there are sixty six different rx types, it is very unlikely that a media interface will be receiving them all and in fact the list could be short, consisting of just a few snapshot summaries.

Note that each time a snapshot command is executed only the current info is processed, which means that if a buffer wraps between snapshot commands the oldest entries are lost. This is acceptable because the snapshot summary is not designed to catch transient packets. Those are displayed in real time by the periodic process and can be captured by the error logging facility.

20.16 Latency Tracer API Functions

The following are the API functions currently used in the Receive Trace Facility (see also Rx Trace Server and Client APIs).

To store an entry onto the FIFO, call the `rx_trace_coalesce()` function.

```
void rx_trace_coalesce (rx_trace_efifo_st *efifo,
                      rx_trace_entry_st *entry);
```

To process request for client services, call the `rx_trace_client_req()` function.

```
void rx_trace_client_req(rx_trace_msg_st *msg,
                        rx_trace_config_st *config,
                        rx_trace_status_st **status);
```

To manage connection id's between modules, call the `rx_trace_coid_manager()` function.

```
int rx_trace_coid_manager(char *media, int *coid);
```

To initialize the config struct passed by the media client, call the `rx_trace_init_config()` function.

```
void rx_trace_init_config(rx_trace_platform_st *platform,
                         rx_trace_config_st *config);
```

To initialize common rx trace parser support, call the `rx_trace_init_parser()` function.

```
void rx_trace_init_parser(void);
```

Note The configuration and status structures are `malloc'd` by the rx trace parser code and `free'd` by the client code.

To monitor and manage the media server's trace buffers, call the `rx_trace_periodic()` function.

```
void rx_trace_periodic(rx_trace_config_st *config,
                      rx_trace_status_st **status);
```

To retrieve an entry from the FIFO, call the `rx_trace_post()` function.

```
rx_trace_entry_st *rx_trace_post (rx_trace_efifo_st *efifo)
```

To pass a buffer server request from the client, call the `rx_trace_server_req()` function.

```
extern void rx_trace_server_req (rx_trace_msg_st *message,
                                 rx_trace_buffer_st **media_array,
                                 rx_trace_entry_st *media_entry,
                                 rx_trace_efifo_st *media_efifo);
```

To write the trace start time, call the `rx_trace_start()` function.

```
rx_trace_entry_st *rx_trace_start (rx_trace_buffer_st *buffer,
                                   ulong timestamp);
```

To write the `stop1` time, call the `rx_trace_stop1()` function.

```
void rx_trace_stop1 (rx_trace_entry_st *entry, paktype *pak);
```

To write the second of two "stop" timestamps, call the `rx_trace_stop2()` function.

```
void rx_trace_stop2 (rx_trace_entry_st *entry, paktype *pak);
```

20.17 Latency Tracer CLI Commands

When the "special" image with the Receive Latency Trace Facility is loaded at the customer site, the customer support engineers can monitor receive interrupt latency by using the rx trace CLI commands. There are three CLI commands specific to rx trace:

- 1 Parser Exec Command
router> **rxtrace media** ...
- 2 Parser Show Command
router> **show rxtrace media** ...
- 3 Parser Test Command
router> **test rxtrace ... media**

20.17.1 Parser Exec Command

The rx trace exec command syntax is as follows:

```
router> rxtrace media threshold [ fastswitch usec ] [ process-switch usec ]
```

The `threshold` option sets the maximum processing time in microseconds for each switching type (fast and process). Packet processing times that exceed these values are considered threshold errors.

```
router> rxtrace media errmsg [ enable ] [ disable ]
```

The `errmsg` option enables or disables real time threshold error messaging to the console. The output is rate limited to one message per second.

```
router> rxtrace media [ entries min max ] [ period millisec ]
```

The `entries` option sets the minimum and maximum number of buffer entries for a given media instance. The actual size is adjusted between these values by the periodic process.

The `period` option sets the minimum sleep period between executions of the periodic process. The actual period is adjusted between this minimum and the default maximum by the periodic process. Note that the maximum period is set at runtime and can't be changed.

```
router> rxtrace media [ clear ] [ reset ]
```

The `clear` option clears the counters of all the `rxtypes` buffers of the selected media.

The `reset` option clears the counters, frees status memory, and sets runtime defaults.

20.17.2 Parser Show Command

The rx trace show command syntax is as follows:

```
router> show rxtrace media [ config ] [ buffers ] [ snapshot ] [ details ]
```

The `config` option displays the global config and some general statistics.

The `buffers` option shows the current buffer stats for each received `rxtypes`.

The `snapshot` option summarizes the current data in the circular trace buffer.

The `details` option displays `ifname`, `rxtypes`, `linktype` and `switching` for each trace.

20.17.3 Parser Test Command

The `test` command syntax is as follows:

```
router> test rxtrace [ client-api ] [ server-api ] media
```

The `server-api` option tests the message request mechanism as well as the published server API functions that are used to manage the trace buffers of the selected media instance.

The `client-api` option tests the functions used to manage the config and status structures. It also creates test scenarios that cause the periodic process to make changes to the trace buffers, thus testing the periodic process.

20.18 Latency Tracer Implementation

There are three levels of implementation for tracing receive packet latency:

- 1 platform specific
- 2 media common
- 3 rx trace library

The platform code is the context for the rx trace facility. At bootup time the platform subsystem initialization function for a particular media calls the routine that creates and initializes the rx trace resources. The implementation example given below is for the AMD Fast Ethernet on the C3600 platform.

20.18.1 Platform Specific Server

The starting point is the platform-media source code file, which contains the C3600 subsystem `init` function for the AMD FE media:

```
pas/if_3600_amdfe.c
```

This source file #includes the platform and media header file which contains the media's receive interrupt handler as an inline function:

```
pas/if_c3600_amdfe.h
```

This header file #includes the common rx trace header file for the AMD FE media type, which has the tracepoint function calls and the macros:

```
pas/if_amdfe_rx_trace.h
```

This is where the trace buffer prototypes are externally declared, and where the "start" and "stop" tracepoint macros are defined. This is also where the compiler switch is used to include or remove the rx trace facility for this media type.

This file also #includes the rx trace server library that defines the tracepoint functions and the server API functions:

```
os/rx_trace_server_api.h
```

Taken together, these files describe the server side of the rx trace facility. Note that the tracepoints must live within the same scope as the receive interrupt handler inline in order to have direct write access to the trace buffers. Thus, the server structures, tracepoints and interrupt handler are all anchored to the subsystem `init` source file.

20.18.2 Platform Specific Client

Starting back at the media's platform subsystem `init` source file:

```
pas/if_c3600_amdfe.c
```

The platform's subsystem `init` function is modified to include the `init` macro that creates an `amdfe rx trace instance`:

```
REG_ADD_AMDFE_RX_TRACE(AMDFE_C3600_RX_TRACE_INFO)
```

These macros are defined in a platform specific rx trace header file, along with the platform defaults for rx trace, such as the maximum and minimum number of trace entries, the maximum and minimum periodic interval, the default threshold, etc:

```
pas/if_c3600_amdfe_rx_trace.h
```

The platform header file then #includes the rx trace client API:

```
os/rx_trace_client_api.h
```

This brings in the client library functions and structures of the rx trace facility. Note that the direction of the dependencies points to the generic from the specific, so changing the platform, for example, does not create dependency errors:

```
pas/if_c3600_amdfe_rx_trace.h -> pas/if_amdfe_rx_trace.h -> os/rx_trace_api.h
```

20.18.3 Implementation File List

The following is a list of files for the rx trace AMD Fast Ethernet implementation:

- New Files:

Rx Trace Library Files

```
os/rx_trace_server.c
os/rx_trace_server_api.h
os/rx_trace_client.c
os/rx_trace_client_api.h
os/rx_trace_exec_options.h
os/rx_trace_show_options.h
os/rx_trace_registry.reg
```

Rx Trace AMDFE Media Instance

```
pas/if_amdfe_rx_trace.c
pas/if_amdfe_rx_trace.h
pas/if_c3600_amdfe_rx_trace.h
```

- Modified Files:

Rx Trace AMDFE Platform Specific

```
h/parser.h
pas/if_c3600_amdfe.c
pas/if_c3600_amdfe.h
```

The following are the modifications to the parser and platform files:

- sys/h/parser.h
 - + PARSE_ADD_RX_TRACE_EXEC, /* exec mode rx trace commands */
 - + PARSE_ADD_RX_TRACE_SHOW, /* show mode rx trace commands */
- pas/if_c3600_amdfe.c
 - + #include "../pas/if_c3600_amdfe_rx_trace.h"
 - static void amdfe_subsys_init (subsys *subsys)
!
! /*
! * If the switch 'RX_TRACE_FACILITY' is defined register
! * the server and startup a client process.
! */
! REG_ADD_RX_TRACE_FACILITY(AMDFE_C3600_RX_TRACE_INFO);
- pas/if_c3600_amdfe.h
 - + #include "if_c3600_amdfe_rx_trace.h"
 - static inline void amdfe_process_receive_packet (...)
!
! rx_pak->rxtyp = ether_decode_encapsulation_inline(rx_pak, ...)
!
! AMDFE_RX_TRACE_START(rx_pak->rxtyp);

The “stop” macros are placed right after the calls to Cisco IOS. There are also a few “stop” macros in the “post-coalesce” functions for process and fast switching.

Note that macros are used so that they will be removed by the compiler if the RX_TRACE_FACILITY compiler switch is not defined.

20.19 Creating a Media RX Trace Instance

There are three modules to a media instance of rx trace:

- 1 Buffer Management Server
- 2 Config Management Client
- 3 Parser Command User Interface

All three are initialized by a create function that is called by the media's subsystem `init` function. The `create` function and the `init` functions live in a new source file specific to the type of media:

```
pas/if_amdfe_rx_trace.c
```

The following are the `create` and `init` function examples that together constitute a complete media instance, in this case for the AMDFE media. Note that there will be other media instances that exist in the same memory address space, such as `amdp2`, `async_cd2430`, and `serial`, so the data structures and functions with names specific to the AMDFE media are declared with the media identifier `amdfe`.

20.19.1 Media Instance Creation

This one function is called by the subsystem `init` function. The `platform` structure is declared in the platform-media header file. The three `init` functions are described below.

```
#include "if_c3600_amdfe_rx_trace.h"

void amdfe_rx_trace_create (rx_trace_platform_st *platform)
{
    amdfe_rx_trace_server_init("amdfe");
    amdfe_rx_trace_client_init("amdfe", platform);
    amdfe_rx_trace_parser_init("amdfe");
}
```

20.19.1.1 Server Instance Initialization

This is where the rx trace server for this media type is anchored in real memory. The server is defined by this media's trace buffer array, its current trace entry and trace entry FIFO. The server must be in the same memory space as the trace points so they can have direct access to the trace buffers.

At runtime the server lists itself with the connection manager and uses the connection id to register its request function so it can receive trace data requests and buffer management commands.

The registry function is invoked as follows:

```
reg_invoke_rx_trace_server_req(coid, message)
```

Creating a Media RX Trace Instance

Passing the connection id for this media selects this server's request function, which then passes its trace buffer array to the library function that knows how to service the message.

```
#include "if_amdfe_rx_trace.h"

rx_trace_buffer_st *amdfe_rx_trace_array [MAX_RXTYPES];
rx_trace_entry_st *amdfe_rx_trace_entry = NULL;
rx_trace_efifo_st amdfe_rx_trace_efifo;

void amdfe_rx_trace_server_req (rx_trace_msg_st *msg)
{
    rx_trace_server_req(msg, amdfe_rx_trace_array);
}

void amdfe_rx_trace_server_init (char *media)
{
    int conid;

    memset(amdfe_rx_trace_array, 0, sizeof(rx_trace_buffer_st) * MAX_RXTYPES);
    memset(amdfe_rx_trace_efifo, 0, sizeof(rx_trace_efifo_st));

    rx_trace_init_nullbuf(amdfe_rx_trace_array);
    rx_trace_conn_manager(media, &conid);
    reg_add_rx_trace_server_req(conid, amdfe_rx_trace_server_req, "amdfe
server");
}
```

20.19.1.2 Client Instance Initialization

This is where the client for this media type is anchored in real memory. The media's periodic process is launched from this memory space, so it has access to the data structures. The parser commands are initialized here for this media type, so user commands generate messages to this client instance.

At runtime the client initializes the config structure with platform defaults. It then asks the connection manager for the media's connection id and uses it to register the client request function used to receive parser exec and show commands.

The registry function is invoked as follows:

```
reg_invoke_rx_trace_client_req(coid, message)
```

Passing the connection id for this media selects this clients's request function, which then passes its trace configuration and status buffer array to the library function that knows how to service the message.

The client then starts up the periodic process, which makes requests for trace data, and looks for threshold errors if message reporting is enabled. It also watches for buffer overwrites and adjusts the buffer size and sleep period accordingly.

```
#include "if_c3600_amdfe_rx_trace.h"

rx_trace_config_st amdfe_rx_trace_config;
rx_trace_status_st *amdfe_rx_status_array[MAX_RXTYPES];
pid_t amdfe_rx_trace_pid;

void amdfe_rx_trace_client_req (rx_trace_msg_st *msg)
{
    rx_trace_client_req(msg, &amdfe_rx_trace_config,
amdfe_rx_trace_status_array);
```

```

}

void amdfc_rx_trace_periodic (void)
{
    rx_trace_periodic(&amdfc_rx_trace_config, amdfc_status_array);
}

void amdfc_rx_trace_client_init (char *media, rx_trace_platf_st *platform)
{
    rx_trace_init_config(platform, &amdfc_rx_trace_config);
    rx_trace_conn_manager(media, &amdfc_rx_trace_config.coid);
    reg_add_rx_trace_client_req(amdfc_rx_trace_config.coid,
        amdfc_rx_trace_client_req, "amdfc client");
    amdfc_rx_trace_pid = process_create(amdfc_rx_trace_periodic, "amdfc
periodic", NORMAL_STACK, PRIO_NORMAL);
}

```

20.19.1.3 Parser Instance Initialization

The rx trace facility appears to the user as the keyword "amdfc" in both the exec and show command line options. The subsequent options selected by the user are passed from the parser instance to the client through a function registry call.

```

rxtrace amdfc threshold usec
rxtrace amdfc entries max min
rxtrace amdfc period millisec
rxtrace amdfc errmsg [ enable | disable ]
rxtrace amdfc clear
rxtrace amdfc reset
show rxtrace amdfc config
show rxtrace amdfc buffers
show rxtrace amdfc snapshot
show rxtrace amdfc details

#define ALTERNATE amdfc_rx_trace_parser_cmd
#include "../os/rx_trace_exec_options.h"

KEYWORD (rx_trace_exec_token, ALTERNATE, NONE,
    "amdfc", "Configure Rx Tracing for AMD Fast Ethernet", PRIV_USER);

LINK_POINT(amdfc_rx_trace_exec_commands, rx_trace_exec_token);
#undef ALTERNATE

#define ALTERNATE amdfc_rx_trace_parser_cmd
#include "../os/rx_trace_show_options.h"

KEYWORD (rx_trace_show_token, ALTERNATE, NONE,
    "amdfc", "Display Rx Tracing for AMD Fast Ethernet", PRIV_USER);

LINK_POINT(amdfc_rx_trace_show_commands, rx_trace_show_token);
#undef ALTERNATE

```

Creating a Media RX Trace Instance

```

static parser_extension_request amdfc_rx_trace_parser_commands[ ] = {
    { PARSE_ADD_RX_TRACE_EXEC, &pname(amdfc_rx_trace_exec_commands) },
    { PARSE_ADD_RX_TRACE_SHOW, &pname(amdfc_rx_trace_show_commands) },
    { PARSE_LIST_END, NULL }
};

#include "if_c3600_amdfc_rx_trace.h"

uint_t amdfc_rx_trace_parser_coid;

void amdfc_rx_trace_parser_cmd (void)
{
    rx_trace_parser_st parser;
    parser.hdr.request = GETOBJ(int,2);
    parser.hdr.reqsize = sizeof(rx_trace_parser_st);
    parser.request = GETOBJ(int,2);
    parser.threshold = GETOBJ(int,3);
    parser.min_period = GETOBJ(int,3);
    parser.max_entries = GETOBJ(int,3);
    parser.min_entries = GETOBJ(int,4);
    parser.enable = GETOBJ(int,3);

    reg_invoke_rx_trace_client_req(amdfc_rx_trace_parser_coid, &parser.hdr);
}

void amdfc_rx_trace_parser_init (char *media)
{
    rx_trace_init_parser();
    rx_trace_conn_manager(media, &amdfc_rx_trace_parser_coid);
    parser_add_command_list(amdfc_rx_trace_parser_commands, "amdfc");
}

```

20.19.2 Copying Another Media Instance

In theory one can copy the files for amdfc to new rx trace media files, say "amdp2", and change every "amdfc" character string to amdp2, and end up with the instance creation code for the new media type. One would do the same for the platform-media file, though the description string may need more changes. For example:

```

cp pas/if_amdfc_rx_trace.c pas/if_amdp2_rx_trace.c
cp pas/if_amdfc_rx_trace.h pas/if_amdp2_rx_trace.h
cp pas/if_c3600_amdp2_rx_trace.h pas/if_c3600_amdp2_rx_trace.h
and then use an editor to change all the occurrences of "amdfc" to "amdp2", and to search for other
changes.

```

Use **sed** to make the changes by creating a script file and running it on the source:

```

sed_script_file

{
    s/amdfc/amdp2/g
    s/AMDFC/AMDP2/g
    s/ AMD / P2 /g
    s/ Fast / /g
}

```

and running it:

```
sed -f sed_script_file pas/if_amdfc_rx_trace.c > pas/if_amdp2_rx_trace.c
```

```
sed -f sed_script_file pas/if_amdfe_rx_trace.h > pas/if_amdp2_rx_trace.h
sed -f sed_script_file pas/if_c3600_amdfe_rx_trace.h
pas/if_c3600amdp2_rx_trace.h
```

After the new files are created and the changes are made one would modify the existing file containing the `subsys_init()` function for the new media on the given platform by including the platform-media header file, and adding the macro that creates the rx trace media instance. For example:

```
pas/if_c3600_amdp2.c

+ #include "../pas/if_c3600_amdp2_rx_trace.h"

--- static void amdp2_subsys_init (subsstype *subsys)
!
! /*
! * If the switch 'RX_TRACE_FACILITY' is defined register
! * the server and startup a client process.
! */
! REG_ADD_RX_TRACE_FACILITY(AMDP2_C3600_RX_TRACE_INFO);
```

Now just put in the tracepoint macros and add `#define RX_TRACE_FACILITY` and there it is, receive packet latency tracing on "amdp2".

20.20 Rx Trace Server and Client APIs

There is a server API and a client API which declare and describe the rx trace library structures and functions. The following is a list of the library objects used by the server and the client instances. Detailed descriptions are in the header files:

- Server Side RX Trace Library Declarations

```
os/rx_trace_server_api.h

rx_trace_buffer_st
rx_trace_efifo_st
rx_trace_msg_st
rx_trace_buffer_msg_st

rx_trace_server_req()
rx_trace_coid_manager()
rx_trace_start()
rx_trace_stop1()
rx_trace_stop2()
rx_trace_coalesce()
rx_trace_post()
```

- Client Side RX Trace Library Declarations

```

os/rx_trace_client_api.h

    rx_trace_platform_st
    rx_trace_config_st
    rx_trace_status_st
    rx_trace_parser_st

    rx_trace_client_req()
    rx_trace_periodic()
    rx_trace_init_config()
    rx_trace_coid_manager()

```

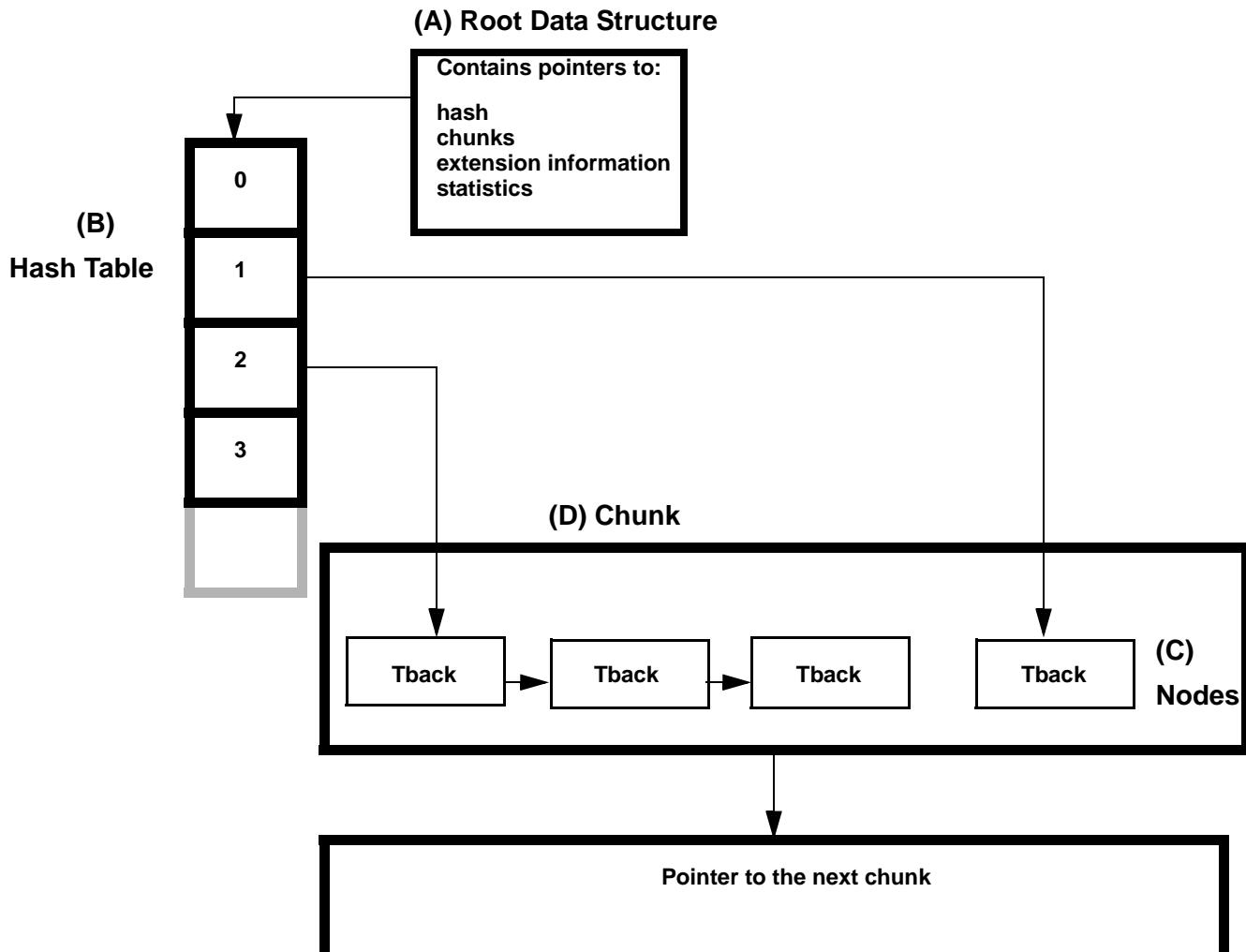
20.21 Traceback Recording

The Traceback Recording API (*new in 12.2T*) provides a means for efficiently storing and retrieving tracebacks. It is useful for gathering tracebacks to aid in post-event debugging. For example, if you have a hash table of events that occurred during a long test run, you can use a traceback to look at the hash table and dig out the events to see how those events got generated. Tracebacks can help give an idea of the codepaths involved, so you can focus on any unusual codepaths that have not yet been considered.

Tracebacks are stored via a hashing mechanism with the following components, shown as (A) through (D) in Figure 20-2:

- (A) is the root data structure that contains pointers to the hash, reference counters, extension information, and so forth.
- (B) is the hash table, indexed by a hash that is calculated from the sum of all the individual program counters (PCs) within a traceback. Each program counter indicates an individual execution point in the code. The anticipation is that (A) will be large enough that collisions between tracebacks will be moderately rare. Hash resizing is not done.
- (C) are the traceback descriptor nodes, named Tback in Figure 20-2. They contain the traceback data, linkage to the next collision node, and an extra hash to speed up comparisons.
- (D) is the large chunk of `malloc`'d memory from which the traceback descriptor nodes (C) are contiguously allocated. When we exhaust one chunk, we allocate another and link to it. As nodes are never individually removed we don't need to worry about coalescing or fragmentation.

Figure 20-2 Hashing Mechanism Components



20.21.1 Traceback Recording List of Functions

The Traceback Recording API includes the following functions in `os/traceback_rec.c`:

- `tb_rec_add()` allocates space within a traceback store and returns a handle to the traceback.
- `tb_rec_delete()` removes all tracebacks and cleans up resources.
- `tb_rec_dump()` displays an individual traceback.
- `tb_rec_new()` reserves space to store tracebacks.
- `tb_rec_print()` displays all tracebacks.

For more information on these functions, see the respective reference pages in the *Cisco IOS API Reference*.

20.21.2 Traceback Recording Implementation

Perform the following steps to implement traceback recording capability, which allows you to gather tracebacks as aids in post-event debugging:

Step 1 Create the hash table in the protocol init code.

For example, add a hash table of size 2^8 with 2000 nodes reserved for use, limit the stack depth to 2, enable counting of duplicate program counters, and enable sorting for ordered output, as follows:

```
root = tb_rec_new(8, 2000, TB_REC_EXTB_32BIT | TB_REC_EXTB_SORTED, 2,
                  TB_REC_ALLOC_PROCESS, "all my tbs", 0);
```

Step 2 Record a traceback, as follows:

```
for (i = 0; i < 10000; i++) {
    tb_rec_add(root, &a_tb, 0);
}
```

Step 3 Add the code to display the traceback that you have a handle on, as follows:

```
tb_rec_print(root, a_tb, "> ", "\n\n", (tb_rec_printer)printf);
```

This is the means to print an individual traceback. For example, you can call this function from your client code in response to a CLI when **show myprotocol traceback x** is entered. The following is an example of the output:

```
> 0x602BE308 0x602BE934 0x602C14D8 ... (1 seen)
```

Step 4 Add the code to display all tracebacks recorded along with detailed statistics, as follows:

```
tb_rec_dump(root, "# ", "\n", "> ", "\n", TRUE, (tb_rec_printer)printf);
```

This is useful when you want to see all the tracebacks that were recorded. The following is an example of the output:

```
> [4] 0x602BE308 0x602BE934 0x602C14D8 ... (1 seen)
> [3] 0x602BE308 0x602BE934 0x602C5FE4 ... (2 seen)
> [2] 0x602BE9EC 0x602C49B4 0x602C573C ... (1 seen)
> [1] 0x602BE308 0x602BE990 0x602C49B4 ... (1 seen)
# 4 unique tracebacks recorded in 20 nodes, using 640 bytes
```

Step 5 Add the code to destroy all held tracebacks, as follows:

```
tb_rec_delete(root);
```

This will free up space, for example if a feature is disabled.

To get an idea of how to start using the Traceback Recording API, see `tb_rec_example` at the end of the `os/traceback_rec.c` file. For more documentation on the Traceback Recording API, see the `h/traceback_rec.h` file.

Note See `bugtrace()` as an example of how to generate immediate tracebacks using `buginf()`.

20.22 Memory Traceback Recording

The Memory Traceback Recording feature (*new in 12.3T*) uses the services provided by the “Traceback Recording” feature to enable users to obtain the *entire* traceback associated with the allocation of a memory block. The program counter that caused the memory allocation is stored in the `allocator_pc`. Currently in IOS, for example, when a memory block is allocated, `allocator_pc` is stored in the block. With the stored information, IOS currently allows users to gather statistics on how much memory has been allocated by `allocator_pc`. However, there are several cases where it is a common piece of code that makes the memory allocation on behalf of other clients. As a result, all blocks thus allocated have the same `allocator_pc`, while the actual users of the blocks are not known. On such cases, once the `allocator_pc` debugging is not possible, one tends to look into the process summaries. However, this can be extremely misleading in cases where a process uses, for example, the `reg_invoke()` function to notify other components about events; the handlers of the `reg_invoke()` function may in turn allocate memory. The process that used the `reg_invoke()` function ends up being blamed. Memory Traceback Recording, in this case, because it records the entire traceback associated with a memory allocation, is helpful by providing more information on the origins of the block.

For more information on Memory Traceback Recording, please see EDCS-314190 “Memory Traceback Recording Software Functional Specification” at the following URL:

http://wwwin-eng.cisco.com/protected-cgi-bin/edcs/edcs_attr_search.pl

The following URL is also helpful:

http://wwwin-eng.cisco.com/Eng/IOSTech/IOSInf/SW_Specs/mem_traceback_event_recording_user_guide.doc

20.23 TCL

TCL (Tool Command Language) is widely used as a scripting language in the testing community. If your image has TCL 8.3.4 (*New in 12.3T*) and you want to write a CLI interactive “wizard” or something like that, we have a TCL package authored for that purpose. The package allows you to write custom character-based menus and wizards via a simple ASCII flat file. For example, if you want to guide a user through a router configuration, your script can prompt the user with as many sentences of explanation as you wish for each parameter, one at a time (optionally providing valid choices or ranges). When the user has responded to all the prompts, your can configure the router, instead of having the user type in a mile-long command line with one hand on the reference manual.

If you are interested, you can find the package here:

http://wwwin-eng.cisco.com/Eng/IOSTech/WWW/Infrastructure/Serviceability/tcl/tcl_index.htm

The TCL 8.3.4 code also has built-in UTF character encoding support. And it has support for other character encodings if you configure the router to point to the external encoding file. Examples of calls you can use are:

```
Tcl_ExternalToUtfDString(. . .)
Tcl_ExternalToUtf(. . .)
Tcl_UtfToExternalDString(. . .)
Tcl_UtfToExternal(. . .)
```

20.24 Embedded Syslog Manager

The new ESM (Embedded Syslog Manager, formerly called SEP) (New in 12.3T), provides a programmable framework that allows you to escalate, filter, correlate, route, and customize (augment, tag, etc.) syslog messages prior to delivery by the Cisco IOS logger. ESM allows you to write custom syslog filters to post-process syslog messages prior to distribution. ESM has extensions that allow you to create new syslog messages as well.

ESM filters are written in TCL (your image must have TCL 8.3.4). For more info on ESM, see the “Embedded Syslog Manager Feature Description” at:

http://wwwin-eng.cisco.com/Eng/IOSTech/WWW/Infrastructure/Serviceability/syslog/documents/ESM_EFT.doc

For more information on the ESM program, see:

http://wwwin-eng.cisco.com/Eng/IOSTech/WWW/Infrastructure/Serviceability/syslog/sep_index.htm

20.25 How to Capture Console Output

This section describes a few different ways to capture console output.

To capture console output (including `printf()` output) that can be sent elsewhere for debugging, complete the following steps:

Step 1 Use the `buffer_stdout()` and `restore_stdout()` functions that are defined in `tty/tty_redirect.c`:

```
void *buffer_stdout (char *buffer,
                     uint buffer_size,
                     buffer_type buf_type);

uint restore_stdout (void *handle);
```

The `buffer_stdout()` function differentiates which process generates the characters.

The `buffer_stdout()` function differentiates what the destination of the characters is (stdout or stderr).

The user just needs to specify a character buffer with its length and circular/linear type, and the `buffer_stdout()` function takes care of the buffer management, etc.

Step 2 Allocate a buffer and pass it to the `buffer_stdout()` routine, which redirects the TTY of the process that is executing the call to `buffer_stdout()`.

Step 3 When you are finished, restore the stdout using the `restore_stdout()` function.

To redirect all console output, irrespective of which process generates the output and whether the output was sent to stderr or stdout, complete the following steps:

Step 1 Use the `platform_redirect_console_output()` and `platform_reenable_console_output()` functions:

```
#include "ttysrv.h"
void platform_redirect_console_output(void (*fn) (uchar));
void platform_reenable_console_output(void);
```

These functions are used for implementing high availability redundancy when redirecting the output of the standby Route Processor (RP) to the active/primary RP.

The `platform_redirect_console_output()` function works at a much lower level than the `buffer_stdout()` function; for instance, each character that is sent to the console driver to be printed will be sent to the alternate function specified by the caller of the `platform_redirect_console_output()` function.

The `platform_redirect_console_output()` function does not differentiate which process generates the characters.

The `platform_redirect_console_output()` function does not differentiate what the destination of the character is (`stdout` or `stderr`).

The caller of the `platform_redirect_console_output()` function must take care of the buffer management, etc.

Step 2 Use `platform_redirect_console_output()` to redirect the console output.

Step 3 Use `platform_reenable_console_output()` to re-enable console standard output.

20.26 Cisco Error Number (cerrno)

The Cisco Error Number, `cerrno`, is a structured error return code that can be used to indicate a globally unique subsystem-specific error. Such an error code substantially helps increase the specificity of the information passed to the user in the event of an error. Also, the subsystem-specific errors uniquely translate to the error strings in the system. With this mechanism, the error information can accurately be tunneled through a layered software system.

For example, consider the following scenario: the application calls the `foo()` function, which in turn calls the `bar()` function, and both of them can return `cerrno`. If the call to `bar()` fails and returns a `cerrno` value, the `cerrno` value returns to the application. Thus, the outermost layer (the application) has the ability of knowing what the exact error is, what caused it, and how severe it is, as all of this information is captured in the `cerrno` return code that is returned by `bar()` and tunneled up.

The structured error return value provides three major pieces of information:

- The severity of an error (fatal, warning, no error).
- The subsystem that caused the error (explained further in 20.26.1.3.1 “Error Code String Mappings”).
- The subsystem-specific error code (explained further in 20.26.1.3.1 “Error Code String Mappings”).

The error return value is a 32-bit integer. This integer is divided up as follows:

XYZTSSSSSSSSSSSSSEEEEEEPPPPPPPPP

XYZ	The error class (indicates the severity of the error).
T	Third party subsystem (if set).
SSSSSSSSSSSS	The subsystem number that generated the most recent error. [13 bits = 8192 values]
EEEEEE	The subsystem specific error number. [6 bits = 64 values]
PPPPPPPP	The pass-thru POSIX error code. [9 bits = 512 values]

The error class encodings are:

XYZ

000 - CERR_CLASS_SUCCESS

001 - CERR_CLASS_INFO

010 - CERR_CLASS_WARNING

```

011 - reserved
100 - CERR_CLASS AGAIN
101 - CERR_CLASS FATAL
110 - CERR_CLASS RESOURCE
111 - reserved

```

Note An error return value of zero means success. There are only 64 subsystem-specific error values available (in addition to the POSIX bits). If this is not sufficient, a subsystem may use multiple subsystem numbers to define additional values.

20.26.1 Usage Guidelines

The following sections describe how to use `cerrno`. The sections discuss the two parts of the `cerrno` API. Furthermore, the structures that make up the `cerrno` information are presented. The macros and the decoding API are also mentioned.

The sections are divided in the following way:

- Overview
- Functional Structure and Data Structures
- Interface Description

20.26.1.1 Overview

The `cerrno` API consists of two parts:

- `cerrno` encoding—this refers to the process of formatting the globally unique (system-wide) 32-bit entities, which are then used by the functions to return an error.
- `cerrno` decoding—this refers to the process of translating the unique 32-bit error code to a corresponding error string.

The `cerrno` encoding API is a set of macros that format the error codes for various categories of error.

20.26.1.2 Functional Structure and Data Structures

The format of the meta-data (error code to string mapping information), provided by the clients that generate the `cerrnos`, are kept the same across OSes, so that the `cerrno` mechanism can be used seamlessly when Cisco IOS application code is migrated across OSes. The mechanism by which the provided meta-data is packaged/discovered at run-time, or made available to the decoding mechanism may be different across the OSes. For example, in Cisco IOS, the meta-data file needs to be included as part of the client subsystem.

The meta-data provided by the clients that generate the `cerrno` is based on the following structures:

```

typedef struct cerr_description {
    int subs_error_code; /* Subsystem-specific error code. */
    const char *subs_error_string; /* String translation of error code.
                                    (Max. size of string is 250) */
} cerr_description_type;

typedef struct cerr_dll_data_ {
    uint32 version; /* Version of the data structure that is being used. */
}

```

```

uint32 flags; /** Unused */
char *subsystem_name; /** Name of the subsystem that this info pertains to.
                      (Max. size of string is 50) */
uint32 subsystem_number; /** Subsystem number this info pertains to */
uint32 count; /** Number of (error code, string) tuples */
const struct cerr_description *errs; /** Array of structures containing
                                      (error code, string) tuples */
uint32 dll_major_version; /** The major version number of the error code
                           mapping/dll */
uint32 dll_minor_version; /** The minor version number of the error code
                           mapping/dll */
uint32 unused[2]; /** unused */
} cerr_dll_data_type;

```

The above structures need to be statically initialized in a separate .c file for every client that uses a unique subsystem ID to generate the `cerrno`. All clients need to use the same `cerr_data` variable name for the variable of `cerr_dll_data_type` type.

20.26.1.3 Interface Description

The `errno` API is defined in `xos_cerrno.h` and includes:

- Macros for generating the `errno` values for the various categories of error—such as `CERR_V_INFO`, `CERR_V_WARNING` and `CERR_V_FATAL`.
- Macros to test for absolute success (`CERR_IS_OK`), any failure (`CERR_IS_NOTOK`) and macros to test for specific class conditions (such as `CERR_IS_SUCCESS` and `CERR_IS_INFO`)
- Decoding functions:
 - `cerr_strerror (errno)`—translates a given `errno` into its corresponding formatted error string.
 - `cerr_getstrings (errno, cerr_strings_st**)`—translates a given `errno` into its constituent parts.

All decoding functions are thread-safe and reentrant.

20.26.1.3.1 Error Code String Mappings

Any component that returns a `errno` should provide a mapping between the subsystem-specific error codes that are used and their corresponding error strings (referred to as the meta-data file above). The definition of the `errno` and the string mappings are to be provided with the following (for every `errno` subsystem):

- An optional header file that defines the return codes, specific to the component (`errno` subsystem). This file can also define/compose the `errnos` to be returned by the subsystem.

Note This header file itself can be private to the client (that is, the subsystem defining it) and need not be made public, as long as the client does not expect the users of its API to check for specific values of the `errnos` being returned.

-
- A .c file that defines the string mappings between the return codes from the component to their string translations, in an array of `cerr_description_type` struct. This file also provides the encapsulating `cerr_dll_data_type` struct, providing information such as subsystem name

and version info. The name of the variable of `cerr_dll_data_type` type must be `cerr_data`. This .c file should include the `xos_errno_init.h` file. This .c file is not allowed to contain any other data or functions.

Note Whenever the `errno` information for the component is updated, the version number fields in the `cerr_data` structure *must* be incremented appropriately. For example, when adding new `errnos`, removing `errnos` or changing the string translation for an existing `errno`, it is up to the component developer to decide whether the major and/or minor version number is to be incremented (at least one of these must be incremented).

Shown below are sample files, both for the header file defining the `errno` and the .c file defining the error code string mappings:

```
/*
 * xos_cerr_errno.h
 * -----
 * XOS Cerrno Infra -- cerrnos definitions header file
 *
 * April 2008, Selva Subramanian
 *
 * Copyright (c) 2008 by cisco Systems, Inc.
 * All rights reserved.
 * -----
 */
#ifndef __XOS_CERR_CERRNOS_H__
#define __XOS_CERR_CERRNOS_H__
#include "xos_errno.h"
/*
 * Please note that this file providing the errno/ return code definitions
 * can be private, to the client, as long as the APIs returning these Cerrnos
 * do not expect the clients to check for a specific values of these Cerrnos.
 */
/* subsystem id for Cerrno Infra */
#define XOS_CERR_SUBSYS_NUM 0x002
/* subsystem name for Cerrno Infra */
#define XOS_CERR_SUBSYS_NAME "XOS Cerrno Infra"
/* Return codes used by the Cerrno Infra */
typedef enum xos_cerr_rc_enum {
XOS_CERR_RC_ERR_INVALID_ARGS = 0x01,
XOS_CERR_RC_ERR_CORRUPT = 0x02,
XOS_CERR_RC_ERR_ALLOC = 0x03,
XOS_CERR_RC_ERR_SHM_CORRUPT = 0x04,
XOS_CERR_RC_ERR_SHM_ALLOC = 0x05,
XOS_CERR_RC_ERR_LOCKING = 0x06
} xos_cerr_rc_enum_type;
/* Cerrnos returned by the Cerrno Infra */
#define XOS_CERR_CERRNO_INVALID_ARGS
\
CERR_V_FATAL(XOS_CERR_SUBSYS_NUM, XOS_CERR_RC_ERR_INVALID_ARGS,
EINVAL)
#define XOS_CERR_CERRNO_CORRUPT
\
CERR_V_FATAL(XOS_CERR_SUBSYS_NUM, XOS_CERR_RC_ERR_CORRUPT, EINVAL)
#define XOS_CERR_CERRNO_ALLOC \
CERR_V_RESOURCE(XOS_CERR_SUBSYS_NUM, XOS_CERR_RC_ERR_ALLOC, ENOMEM)
#define XOS_CERR_CERRNO_SHM_CORRUPT
```

```
\Date printed: 9/17/2008 Cross-OS Basic OS Infra Software Design
Specification: EDCS-673344
Copyright 2007 Cisco Systems 41 Cisco Highly Confidential - Controlled Access
A printed copy of this document is considered uncontrolled. Refer to the
online version for the controlled revision.
CERR_V_FATAL(XOS_CERR_SUBSYS_NUM, XOS_CERR_RC_ERR_SHM_CORRUPT,
EINVAL)
#define XOS_CERR_CERRNO_SHM_ALLOC
\
CERR_V_RESOURCE(XOS_CERR_SUBSYS_NUM, XOS_CERR_RC_ERR_SHM_ALLOC, EOK)
#define XOS_CERR_CERRNO_LOCKING
\
CERR_V_RESOURCE(XOS_CERR_SUBSYS_NUM, XOS_CERR_RC_ERR_LOCKING, EOK)
/* Macro to include posix error code with Cerrno */
#define XOS_CERR_INC_POSIX_WITH_CERRNO(cerr, posix) (cerr | posix)
#endif /* __XOS_CERR_CERRNOS_H__ */
/*
*-----
* XOS Cerrno Infra -- Cerrnos definitions
*
* Feb 2008, Selva Subramanian
*
* Copyright (c) 2007-2008 by Cisco Systems, Inc.
* All rights reserved.
*-----
*/
#include "xos_cerr_cerrnos.h"
/*
* The following file **MUST** be included, for
* registering mappings from this client.
* This file just needs to be included once from
* this file (providing the string mappings for the
* cerrno) and should not be included elsewhere.
*/
#include "xos_cerrno_init.h"
Date printed: 9/17/2008 Cross-OS Basic OS Infra Software Design
Specification: EDCS-673344
Copyright 2007 Cisco Systems 42 Cisco Highly Confidential - Controlled Access
A printed copy of this document is considered uncontrolled. Refer to the
online version for the controlled revision.
/*
* Following structure provides mapping of return codes from this
* subsystem to their string translations.
* The structure below must be qualified as a "static const"
*/
static const struct cerr_description xos_cerr_table[] = {
{XOS_CERR_RC_ERR_INVALID_ARGS,
"Invalid args"},
{XOS_CERR_RC_ERR_CORRUPT,
"Corrupt data structure"},
{XOS_CERR_RC_ERR_ALLOC,
"Could not allocate from heap"},
{XOS_CERR_RC_ERR_SHM_CORRUPT,
"Corrupt data structure in shared memory"},
{XOS_CERR_RC_ERR_SHM_ALLOC,
"Could not allocate from shared memory"},
{XOS_CERR_RC_ERR_LOCKING,
"Could not lock or unlock mutex"}
```

```

};

/*
 * The name of this variable (of cerr_dll_data_type) must be "cerr_data",
 * and must use the qualifier XOS_CERR_DEFINE.
 * The struct has a major and minor version number fields. These fields
 * **MUST** be changed whenever there is a change in the cerrno to string
 * translation mappings provided above: i.e. when adding new cerrnos
 * to this subsystem, removing cerrnos, or changing the string translation
 * for an existing cerrno. It's up to the component, to decide whether
 * major and/or minor version number will be changed, but note that at least
 * one of this **MUST** be changed.
*/
XOS_CERR_DEFINE cerr_dll_data_type cerr_data = {
CERR_DLL_VER,
0,
XOS_CERR_SUBSYS_NAME,
XOS_CERR_SUBSYS_NUM,
/* Note that xos_cerr_table[0] should have been
 * statically allocated and never be null
 */
sizeof(xos_cerr_table) / sizeof(xos_cerr_table[0]),
xos_cerr_table,
1, /* The major version number of the DLL. */
0, /* The minor version number of the DLL. */
{ 0, 0 }
};

```

20.27 Debugging ASLR Enabled Cisco IOS Images

Many versions of Cisco IOS now support address space layout randomizations (ASLR). ASLR randomly arranges the positions of key data areas. Within Cisco IOS, random data offset (RDO), random text offset (RTO), random heap offset (RHO), and random iomem offset (RIO) are implemented. Traditionally, on every boot, the text (executable) and the data segments are placed in the same location. With the addition of ASLR, the location of the text (executable) and the data segments changes on every boot, and additional procedures are required for debugging. This section describes the new procedures.

With ASLR enabled, addresses listed in the symbols file are not the same as the addresses used by the running image. The addresses in the symbols file are referred to as link-time addresses (LTA) and the addresses on the running image as run-time addresses (RTA). The difference between these addresses is referred to as the random offset.

There is little impact on the day-to-day debugging with ASLR. GDB automatically adjusts for ASLR with interactive sessions, core dumps, and crashinfo files. Tracebacks are automatically adjusted as if there is no RTO. Only when examining the raw output and comparing tracebacks to the GDB using the **backtrace** command, you have to consider the effects of ASLR.

20.27.1 ASLR and Impact on Debugging

There is minimal impact to day-to-day debugging because of ASLR. GDB will automatically adjust for ASLR with interactive sessions, core dumps, and crashinfo files. Tracebacks are automatically adjusted as if there is no RTO. Only when examining the raw output and comparing the tracebacks to the GDB backtrace might a developer have to consider the effects of ASLR.

20.27.2 Determining the RDO and RTO offset

It is possible to determine the random offset of RTO and RDO by comparing the output of the **show region** command on the router with the symbols file. In the symbols file, for a given image, there are the symbols `_start` and `_fdata`, along with their non-offset addresses. The output of the **show_region** command will give a `main:text` and a `main:data` region which correspond to the `_start` and `_fdata` symbols respectively, along with the offset included in the reported addresses. In order to determine the random offsets, the appropriate symbols address from the symbols file must be subtracted from the address of the start of the corresponding region as reported by **show region** command. The below table summarizes this procedure:

Region	Symbol	Offset
<code>main:text</code>	<code>_start</code>	= RTO
<code>main:data</code>	<code>_fdata</code>	= RDO

20.27.3 Debugging ASLR enabled images

20.27.3.1 Debugging RHO and RIO

Even without ASLR, the locations of data in the heap and iomem are expected to be unpredictable from boot to boot. Because of this, RHO and RIO has no effect on the debugging procedures for these memory areas.

20.27.3.2 Debugging console output of RTO and RDO enabled images

There are many sources of memory information from the console. These include debug statements, show commands, tracebacks, and system logs. For some of these, it is preferable to display the link-time address (LTA) while other times it is preferable to display the RTA. This is a decision of the code owners as to which is preferable. To distinguish LTA from RTA, Cisco IOS code prints all LTAs with a `z` appended to the end of the address. For example, a traceback is printed as an LTA, as shown below:

```
-Traceback= 0x607F20E8z 0x6084B164z 0x6084B134z 0x6084DD24z 0x60831248z
```

For addresses that are RTA, any memory reference to text or data symbols include the respective RDO or RTO offset as calculated in [Determining the RDO and RTO offset on page 73](#). To determine a symbol associated with a memory address, the offset must be subtracted before looking for it in the symbols file. Fortunately, tools like RSYM and GDB have been modified to automatically do this adjustment.

Sections that are not listed in the symbols file (like the heap) are not adjusted by either RDO or RTO. Memory references in these sections do not need the RDO or RTO offset to be subtracted.

20.27.3.3 Rsym modifications for ASLR

RSYM is a tool to determine text symbols from tracebacks. It is modified to detect when a symbols file is associated with a RTO image and handle the addresses accordingly. For more information, see Modifications of rsym to support ASLR SFS([EDCS-495619](#)). Three flags have been added to rsym:

- `-t <start>` - Supplies the address of text start. This is the text start value of the **show region** command. If not specified, the text start from the symbols file is used; which is equivalent to no ASLR offset. If specified, any conflicting text start addresses in the file-to-parse is ignored.
- `-z` - Considers all addresses to be LTA even if they are not LTA formatted.

- -R - Prints symbol start addresses adjusted to be RTA.

For non-RTO images, no changes in rsym usage is required. For RTO images where the file-to-parse contains the text start, no changes in rsym usage is required.

If the data-set contains both LTA and RTA, the text offset is not available in the file-to-parse, or RSYM is in interactive mode, the text start needs to be specified with the -t option. This can be either extracted from the output of **show region** command, or the TEXT_START line in a crashinfo file. For more information, see [RTO, RDO and Crashinfo, and GDB on page 74](#).

If the data-set is all LTA (all end in z) the -z flag can be used even if the text start is not available.

Sometimes it is convenient to convert a LTA to a RTA. For example, when running GDB on an image, the RTA is useful. The -R flag can be used to do this conversion.

20.27.3.4 RTO, RDO, Core Dumps, and GDB

Current Cisco IOS releases of GDB support ASLR. GDB automatically detects data and text offset and adjusts symbols to reflect these offsets.

Since the tracebacks printed by the router subtract the RTO offset, it is important to note that these are not consistent with GDB's **backtrace** command. The rsym -R flag can be used to convert these tracebacks.

20.27.3.5 RTO, RDO and Crashinfo, and GDB

Crashinfo has been augmented to print data and text offsets after a register dump, as shown below:

```
TEXT_START : 0x6000E1B4  
DATA_START : 0x629297E0
```

The above information is equivalent to the main:text and main:data as mentioned in [Determining the RDO and RTO offset on page 73](#). With this information, you can do manual debugging as described in [Debugging console output of RTO and RDO enabled images on page 73](#) and [Rsym modifications for ASLR on page 73](#).

GDB currently supports targeting crashinfo files as if they are sparse core dumps. Since it already supports ASLR, no manual memory calculations are required. You can target a crashinfo file with the following command:

```
target cisco-crashinfo <crashinfo-filename>
```

20.28 The sdec Tool and Stack Corruption Troubleshooting

You might be able to use the **sdec** tool created by Preston Chilcote (pchilcot) to troubleshoot stack corruption. To use this tool, follow these steps:

- Step 1** Find Out If There Has Been Stack Corruption
- Step 2** Check the Stacks
- Step 3** Decode
- Step 4** Find Out What Causes a Stack to Overrun
- Step 5** Resolving the Stack Overrun Problem
- Step 6** Find CDETS Defects with Similar Problems

Step 7 Find Out How Much Stack a Function is Allocating

20.28.1 Find Out If There Has Been Stack Corruption

Look in the crashinfo file to find out if there has been stack corruption. Look for the following common indications:

- No traceback. For example:

```
===== Stack Trace=====
```

```
-Traceback=
```

- Only one address in the traceback (which probably doesn't decode to anything). For example:

```
===== Stack Trace=====
```

```
-Traceback= 0x41A6D1B4
```

- Process or interrupt stack is completely full. For example, CSCse81684.

```
s8: 00000002 sp: 46CB30C0 PC: 00000000 ra: 4252A2DC
```

```
- Current Process Stack (0x2EE0 bytes used, out of 0x2EE0 available)
```

- Stackflow messages. For example, CSCsg94837:

```
%SYS-6-STACKLOW: Stack for process IP-EIGRP: HELLO running low, 0/6000
```

- Repeating functions in traceback.

```
0x601AC9AC:process_handle_watchdog(0x601ac984)+0x28
0x601A0A6C:signal_receive(0x601a09c4)+0xa8
0x60147170:watchdog_forced_here(0x601470b4)+0xbc
0x605552E8:regmatch(0x60555040)+0x2a8
0x605552A8:regmatch(0x60555040)+0x268
0x605552D4:regmatch(0x60555040)+0x294
0x605552A8:regmatch(0x60555040)+0x268
0x6055536C:regmatch(0x60555040)+0x32c
0x6055526C:regmatch(0x60555040)+0x22c
0x6055526C:regmatch(0x60555040)+0x22c
0x605552D4:regmatch(0x60555040)+0x294
0x605552A8:regmatch(0x60555040)+0x268
```

20.28.2 Check the Stacks

The input to the **sdec** tool is the stack printed in the “Stack Dump” section. This may be a process or interrupt stack. Sometimes, interrupt stacks overflow from one into the other if they are allocated next to each other in memory. Usually the victim interrupt stack is printed in those cases.

20.28.3 Decode

The **sdec** tool uses **rsym** to decode every available address in the given file. It does its best to filter out functions that it knows aren't valid PCs, but it can't filter them all. That means that the output of **sdec** isn't nearly as reliable as a decode of a traceback generated directly from the router. It is intended to find general patterns of what was happening at the time of the crash.

How to decode?

Use the **sdec** tool in `/users/pchilcot/bin`, which takes a file with the stack, a symbols file, and a text offset if necessary for ASLR images.

```
sdec file_to_decode symbols_file [TEXT_START]
```

Copy and paste the full stack into a file, then run:

```
% /users/pchilcot/bin/sdec file_with_stack symbols_file
```

For example:

```
% /users/pchilcot/bin/sdec temp
/release/124/sym/124-6.T//c2800nm-advipservicesk9-mz.124-6.T.symbols.gz
```

And the example output is:

```
0x41D14C00:lapb_debug_all(0x41d14c00)+0x0
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190afd8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_13_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190afd8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
```

```
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190afd8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190afd8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
```

```

0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190af8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190af8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190af8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788

```

0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190afd8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc
0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190afd8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_l3_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x431C38F8:mv64340_ge_safe_start(0x431c320c)+0x6ec
0x431C33B8:mv64340_ge_safe_start(0x431c320c)+0x1ac
0x40385170:pak_enqueue(0x40385074)+0xfc

```

0x4193A760:ip_ether_macaddr(0x4193a4cc)+0x294
0x40386674:holdq_enqueue(0x403865a4)+0xd0
0x41410000:tvc_qos_setup(0x4140ffdc)+0x24
0x419216B8:ipbuild_head(0x4192150c)+0x1ac
0x400D13EC:ru_account_memory_ro_account(0x400d13a0)+0x4c
0x41922EF0:ipwrite_allow_zero(0x41922d9c)+0x154
0x42310A90:classify_packet(0x4230ee0c)+0x1c84
0x400D15AC:memory_ro_account_chunk_free(0x400d1580)+0x2c
0x4190C1B8:ip_process_pak_internal(0x4190af8)+0x11e0
0x400C5840:free(0x400c574c)+0xf4
0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A63430:ipnat_13_fixup(0x41a626f4)+0xd3c
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4
0x41A66E38:ipnat_translate_before_routing(0x41a65900)+0x1538
0x41A65198:ipnat_in2out_translate(0x41a650cc)+0xcc
0x403B5490:if_dequeue_common(0x403b53f4)+0x9c
0x403856B4:pak_dequeue(0x4038561c)+0x98
0x40386CB4:holdq_dequeue(0x40386bb4)+0x100
0x419752D0:ipcachel_update(0x41974520)+0xdb0
0x419244C8:ipsendnet(0x41924480)+0x48
0x4190D1F0:ip_forward_to_net(0x4190d0e4)+0x10c
0x4190EC50:ip_forward(0x4190d4d8)+0x1778
0x42522980:mgd_timer_set_exptime_internal(0x42522880)+0x100
0x4190C1B8:ip_process_pak_internal(0x4190af8)+0x11e0
0x42522EB0:mgd_timer_set_exptime_common(0x42522e38)+0x78
0x4190C700:ip_process_pak(0x4190c698)+0x68
0x42529094:process_quantum_is_expired(0x42529064)+0x30
0x4252A2CC:process_wait_for_event(0x4252a17c)+0x150
0x4190C7BC:ip_process_input(0x4190c724)+0x98
0x4190C7A8:ip_process_input(0x4190c724)+0x84
0x4190C8F0:ip_input(0x4190c858)+0x98
0x4190C8F0:ip_input(0x4190c858)+0x98
0x42529094:process_quantum_is_expired(0x42529064)+0x30
0x4190C960:ip_input(0x4190c858)+0x108
0x4190C9B4:ip_input(0x4190c858)+0x15c
0x42518310:r4k_process_dispatch(0x425182f4)+0x1c
0x425182F4:r4k_process_dispatch(0x425182f4)+0x0

```

In the above stack, notice the repeating pattern (in bold) of the following functions:

```

0x41A551C4:ipnat_reassemble_payload_pak(0x41a55050)+0x174
0x41A556F0:ipnat_reassemble_payload_pak(0x41a55050)+0x6a0
0x41A61F30:ipnat_handle_skinny_fragments(0x41a617a8)+0x788
0x41A620E4:ipnat_handle_skinny(0x41a61fc4)+0x120
0x41A52EE8:ipnat_open_door(0x41a52e28)+0xc0
0x41A63430:ipnat_13_fixup(0x41a626f4)+0xd3c
0x41A6A0FC:ipnat_find_node(0x41a69fd8)+0x124
0x41A65090:ipnat_out2in_translate(0x41a64fcc)+0xc4

```

This suggests that the logic of NAT is either incorrect, or extremely inefficient.

20.28.4 Find Out What Causes a Stack to Overrun

What causes a stack overrun?

The size of the stack is determined at the time when the process is created. It's an argument to the `process_create()` function:

```
/*
 * managed_chunk_init
 * Initialize the managed chunk manager variables and queue
 */
void managed_chunk_init(void)
{
    managed_chunk_queue = create_watched_queue("Chunk Manager", 0, 0);
    /*
     * Start process to manage chunk growth, for managed chunks
     */
    managed_chunk_pid = process_create(managed_chunk_process, "Chunk
Manager", NORMAL_STACK, PRIO_CRITICAL);
```

The sizes are defined on a per CPU basis:

For example, from `cisco.comp/target-cpu/include/mips/r4k/cpu_4k.h:2571:`

```
/*
 * Process level stack parameters. Used in calls to process_create().
 */
typedef enum stack_size_t {
    SMALL_STACK      = 3000 + IMPL_SLOP,          /* bytes */
    NORMAL_STACK     = 6000 + IMPL_SLOP,          /* bytes */
    CRYPTO_PUBKEY_STACK = 8000 + IMPL_SLOP,        /* bytes */
    MODERATE_STACK   = 9000 + IMPL_SLOP,          /* bytes */
    LARGE_STACK       = 12000 + IMPL_SLOP,         /* bytes */
    HUGE_STACK        = 24000 + IMPL_SLOP,         /* bytes */
    GIANT_STACK       = 60000 + IMPL_SLOP,         /* bytes */
    MEGA_STACK        = 100000 + IMPL_SLOP          /* bytes */
```

It is up to the process to make sure it doesn't use too much stack. Sometimes a function or series of functions get caught in a loop, which is always a software problem. Other times, it might be inefficient use of the space on the stack, such as storing too much data on the stack. This is a compiler or programming inefficiency problem (also always a software problem).

20.28.5 Resolving the Stack Overrun Problem

There are a few common ways to resolve the problem:

- Stop infinite loops, either recursive functions or functions with incorrect logic.
- Optimize functions to use less stack space.
- Increase the size of the stack. This would be done if the above specified two suggestions are completed.

20.28.6 Find CDETS Defects with Similar Problems

Table 20-8 provides a sample of defects addressing stack problems:

Table 20-8 Some Defects that Address Stack Problems

Defect	Description
CSCsg94837	Crash due to %SYS-6-STACKLOW: Stack for process IP-EIGRP
CSCsa82934	c3825 with crypto HW reloads with EIGRP STACKLOW 0/6000 under stress (sfnt drivers)
CSCsg26139	3845/7200 crypto stack corruption (hifn drivers)
CSCsg60727	Crash due to stack overrun and Dynamic DNS Update Timer Process
CSCsg09208	IOS/IPS: Crash loading v6 signature file
CSCsg44748	Illegal opcode exception crash due to stack overrun caused by RTP
CSCse81684	Watchdog timeout or stack corruption in IP INPUT process
CSCsg22426	NAT/Skinny: Stack overflow with fragmented skinny packets
CSCsg01366	CSM config sync cause stacks to run low and crash router
CSCsg08491	processor pool memory corruption crash on application of crypto map
CSCsi17020	Stack overflow with fragmented skinny packets
CSCsj14204	Bus error crash after upgrading to 12.2(28)SB6
CSCsh31203	Level 1 Interrupt Stack Corruption with Voice T1

20.28.7 Find Out How Much Stack a Function is Allocating

To find how much stack a function is allocating on MIPS platform, you need to disassemble it with **gdb**:

(cisco-6.4-r4k-gdb) **disassem bgp_show_network_detail**

Dump of assembler code for function `bgp_show_network_detail`:

```

0x60993f08 <bgp_show_network_detail+0>: addiu   sp,sp,-360
0x60993f0c <bgp_show_network_detail+4>: sw      ra,356(sp)
0x60993f10 <bgp_show_network_detail+8>: sw      s8,352(sp)
0x60993f14 <bgp_show_network_detail+12>: sw      s7,348(sp)
0x60993f18 <bgp_show_network_detail+16>: sw      s6,344(sp)
0x60993f1c <bgp_show_network_detail+20>: sw      s5,340(sp)
0x60993f20 <bgp_show_network_detail+24>: sw      s4,336(sp)
0x60993f24 <bgp_show_network_detail+28>: sw      s3,332(sp)
0x60993f28 <bgp_show_network_detail+32>: sw      s2,328(sp)
0x60993f2c <bgp_show_network_detail+36>: sw      s1,324(sp)
0x60993f30 <bgp_show_network_detail+40>: sw      s0,320(sp)
0x60993f34 <bgp_show_network_detail+44>: sw      zero,300(sp)
0x60993f38 <bgp_show_network_detail+48>: sw      a0,360(sp)
0x60993f3c <bgp_show_network_detail+52>: sw      a1,364(sp)
0x60993f40 <bgp_show_network_detail+56>: lw      s8,4(a0)
0x60993f44 <bgp_show_network_detail+60>: sw      a2,368(sp)

```

Generally, the first few assembly instructions include an operation to subtract some bytes from the sp (stack pointer). Remember we're subtracting because the stack grows from high memory address to low. The amount being subtracted is the size of the frame for this function.

P A R T 4

Network Services

Binary Trees

21.1 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.

Note Cisco IOS binary tree development questions can be directed to the interest-os@cisco.com and os-infra-team@cisco.com aliases.

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

21.1.1 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, deletions, and searches run in $O(\log n)$ time.

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.

21.1.2 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 for an AVL tree is $O(\log n)$.

Three levels of AVL functionality are available:

- Raw AVL tree manipulation functions
- Wrapped functions
- AVL tree extension to allow duplicates

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 multiple AVL trees for data that must be sorted on more than one key at once.

AVL Duplicate trees are being introduced in order to remove the restriction that AVL trees have, in which multiple elements with the same key value are not allowed on the tree.

21.1.3 Radix Trees

Radix trees are currently used in the BGP, PIM and DVMRP tables. Radix trees lend themselves well to IP, where routing decisions are made by matching not only the route, but also the address mask. The search time for a radix tree is $O(n)$.

21.2 Manipulate RB Trees

21.2.1 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);
```

21.2.2 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);
```

21.2.3 Search an RB Tree

Table 21-1 describes the functions available for searching for nodes in an RB tree.

Table 21-1 Functions for Searching an RB Tree

Search Conditions	Function
Node that exactly matches the specified key value.	<code>treeLink *RBTREESearch(rbTree *T, treeKey key);</code>
Overlapping interval in an RB interval tree.	<code>treeLink *RBTREEIntSearch(rbTree *T, treeKey key);</code>
First node in the tree.	<code>treeLink *RBTREEFirstNode(rbTree *T);</code>
Next node in the tree.	<code>treeLink *RBTREENextNode(treeLink *node);</code>
Maximal node that is less than or equal to a specified key.	<code>treeLink *RBTREEBestNode(rbTree *T, treeKey key);</code>
Node equal to or greater than a specified key.	<code>treeLink *RBTreeLexiNode(rbTree *T, treeKey key);</code>
Node with the largest key.	<code>treeLink *RBTREENearBestNode(rbTree *T, treeKey key);</code>
Node with the largest possible interval key that is less than or equal to the interval specified in the key.	<code>treeLink *RBTREEIntNearBestNode(rbTree *T, treeKey key);</code>
First node on the tree's internal free list.	<code>treeLink *RBTreeGetFreeNode(rbTree *T);</code>

21.2.4 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 protectIt);

boolean RBTreeForEachNodeTilFalse(treeProc proc, void *pdata, rbTree *T,
                                  treeLink *start, boolean protectIt);
```

21.2.5 Retrieve Information about an RB Tree

Table 21-2 describes the functions available for retrieving information about an RB tree.

Table 21-2 Functions for Retrieving Information about an RB Tree

Information	Function
Number of free nodes that are not busy.	<code>int RBReleasedNodeCount(rbTree *T);</code>
Number of free nodes on the tree's internal free list.	<code>int RBFreeNodeCount(rbTree *T);</code>
Whether a node is on the tree's internal free list.	<code>boolean RBTreeNodeDeleted(rbTree *T, treeLink *node);</code>

21.2.6 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);
```

21.2.7 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);
```

21.2.8 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);
```

21.2.9 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);
```

21.2.10 Set Up an RB Tree With a Key That Is Non-32 Bits

To set up an RB tree with a key that is non-32 bits (that is, greater or smaller than 32), call the `RBTreeNon32Bit()` function.

```
void RBTreeNon32Bit (rbTree* T,
                      treeKeyMax keymax,
                      treeKeyCompare keycompare)
```

21.3 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.

AVL Duplicate trees are being introduced in order to remove the restriction that AVL trees have, in which multiple elements with the same key value are not allowed on the tree. AVL Duplicate trees are built upon the functionality of AVL trees. There is no change to any of the implementation of AVL trees and hence the existing AVL functionality remains unaffected.

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 return the pointer back to the beginning of your data structure (this function does not adjust any pointers).

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.

21.3.1 Manipulate Raw AVL Trees

21.3.1.1 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.

21.3.1.2 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);
```

21.3.1.3 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,
                  void *paramptor);
```

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);
```

21.3.1.4 Search an AVL Tree

To search an AVL tree for a specified goal key, use the `avl_search()` function.

```
avl_node_type *avl_search(avl_node_type *top, avl_node_type *goal,
                         avl_compare_type compare_func);
```

21.3.1.5 Remove an AVL Tree

To delete and remove an AVL tree, use the `wavl_remove()` function.

```
void wavl_remove(wavl_handle *handle, int thread, void
                  (*freefunc)(void*));
```

21.3.1.6 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);
```

21.3.1.7 Free AVL Tree Resources

There are two ways to free resources associated with an 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 any AVL tree functions. The second way to free AVL 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.

21.3.2 Manipulate Wrapped AVL Trees

Use wrapped AVLs to index a piece of data on several keys at the same time. While you could maintain multiple trees at the same time, this quickly becomes complicated and cumbersome. The proper solution in IOS is to use wrapped AVL trees.

The method for using WAVL trees is similar to the other binary trees and is:

- Step 1** `#include "../util/wavl.h"` — In addition to prototypes for the `wavl_*`() routines, this file also declares several key data structures, including `wavl_node_type`, and `wavl_handle`.
- Step 2** Create the tree — WAVL trees are created with the `wavl_init()` routine.
- Step 3** Insert nodes into the tree — Nodes are inserted using the `wavl_insert()` routine.

Step 4 Find/Delete nodes from the tree — These operations can occur in any order. However, you cannot find a node in a WAVL tree. Rather, you can only find a node in an AVL tree that is part of an AVL tree. This is because individual nodes only exist on AVL trees. Use the `wavl_search()` routine to find a node on an AVL tree that is part of an AVL tree. Delete nodes from a tree using the `wavl_delete()` function.

Step 5 Destroy the tree — When finished with a tree, it should be destroyed so that its memory can be reclaimed. There are several steps you must take to destroy a WAVL tree: Delete each node from the WAVL tree either using `wavl_delete()` or `wavl_delete_thread()`. If the containing structure for the nodes were allocated using `malloc()`, free said structures using `free()`. Call `wavl_finish()` to free the memory associated with the WAVL tree itself. Failure to follow this procedure will cause a memory leak which will eventually cause the router to crash.

21.3.2.1 Using WAVL Data Structures and Defining Necessary Routines

If a WAVL tree consists of more than one AVL tree (as specified in the `wavl_init()` routine), then the structure containing the `wavl_node_type` structure must contain an array of `wavl_node_type` structures, one for each AVL tree on which the node will exist. In such cases:

- When you call routines such as `wavl_insert()` that take a pointer to a `wavl_node_type`, you must pass the pointer to the first (zeroth) element in said array.
- If the `wavl_walk()` or `wavl_do_walk()` routines will be called on the WAVL tree(s), you may need to provide a `findblock` function when you create the WAVL tree(s). You must do this if:
 - You have more than one AVL tree in your WAVL tree.
 - The `wavl_node_type` member is not the first item in the containing structure.

Note What the `findblock` function must do is, given a `wavl_node_type` pointer, return a pointer to the containing structure, which must have been allocated via `malloc()`. This must happen so that the two routines can place a memory lock on the blocks of memory so that they are not deleted by another process while the tree is being walked.

- To call `wavl_walk()` or `wavl_do_walk()`, the items linked onto the WAVL tree(s) must be allocated via `malloc()`.

Warning Calling these routines when the blocks are allocated in some other manner will crash the router!

Since WAVL trees consist of several AVL trees, each tree can have its own key type and thus must have its own comparison function. Each comparison function is specified as an argument to the `wavl_init()` routine.

21.3.2.2 Initialize a Wrapped AVL Tree

WAVL trees are created using the `wavl_init()` routine with the following syntax:

```
boolean wavl_init(wavl_handle *handle,
                  int num_threads,
                  void *(*findblock)(wavl_node_type *),
                  avl_compare_type cf1, ...)
```

where:

- The *handle* parameter is a pointer to a `wavl_handle_type` structure, which must be created by a call to `malloc()`.
- The *num_threads* parameter indicates how many AVL trees are created, and thus, on how many trees each node exists. Some IOS programmers use 1 for the *num_threads*, so that they can use the WAVL routines rather than the AVL routines.
- The *findblock* parameter is a pointer to a function that takes a `wavl_node_type` structure and returns a pointer to the containing structure. This argument defines, given a node in the tree, how to find the key for that particular node.

There may be circumstances in which nodes on a WAVL tree use nodes of type `as_int`, which have a `wavl_node_type` as the first member. Thus, no function is needed in this type of instance and this parameter can be set to `NULL`.

Note One comparison routine for each of the AVL trees must be defined for each thread in the WAVL tree(s).

You must call the `wavl_init()` function before calling any other wrapped AVL function.

21.3.2.3 Insert a Node into a Wrapped AVL 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. If there is any failure to insert into any of the threads, it does not leave the node inserted into only the threads that were successful.

```
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 specific AVL 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_delete_thread()` 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 with `wavl_delete()` and then reinserting wastes a large number of CPU cycles.

In `/vob/ios/sys/atm/atm_arpserv.c`, there is an example of the use of the `wavl_insert()` routine to add a node into a WAVL tree and its associated AVL tree(s). The syntax of the routine is:

```
boolean wavl_insert(wavl_handle *handle, wavl_node_type *node)
```

where:

- The *handle* parameter is a pointer to a previously allocated handle which has been initialized with `wavl_init()`.

- The *node* parameter points to a `wavl_node_type` member of a previously allocated structure. In the example, the structure is allocated with `malloc()`. Remember, if multiple AVL trees are in the WAVL tree, then the *node* parameter must point to the base of the array of `wavl_node_type` structures in the containing structure.
- The Boolean return value indicates whether or not the insertion was successful. Reasons for failure include duplicate keys, invalid *handle*, and invalid *node*. This code does not check the return code, because it apparently does not care whether or not the node is added to the tree.

Note If you have a multi-AVL WAVL tree and you want to insert a node on only one AVL tree, you may use the `wavl_insert_thread()` routine as well. It has the same parameters and return as `wavl_insert()`, except that it has one additional parameter, the tree # or thread # on which you want the node added. In this case, the *node* parameter must still point to the base of the array `wavl_node_t` structures.

21.3.2.4 Traverse a Wrapped AVL Tree

To traverse (walk) a wrapped AVL tree in the previously initialized context, use the `wavl_walk()` function.

Use the following routine to visit each node until *proc* returns FALSE:

```
boolean wavl_walk(const wavl_handle *handle,
                  int thread,
                  avl_walker_type proc,
                  void *paramptr);
```

Use the next two routines in a for/while loop to visit nodes one at a time:

- 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 *element,
                             int thread);
```

Using the three routines provided here, there are two ways to walk a WAVL tree:

- Use the `wavl_walk()` routine to invoke a specific function (*proc*) on every node in the AVL tree specified by *thread*, until *proc* returns FALSE. When *proc* is invoked, it is passed the pointer to the current node and *paramptr* (presumably data useful to *proc* but opaque to the `wavl_walk()`).
 - If *proc* returns FALSE at any node, no further nodes are visited and `wavl_walk()` returns FALSE. Otherwise, every node in the AVL tree specified by *thread* is visited and `wavl_walk()` returns TRUE.
 - Remember, to use `wavl_walk()`, the nodes on the WAVL tree must be allocated with `malloc()`.
- Use the two routines `wavl_get_first()` and `wavl_get_next()` in a loop. This allows the processing of each node to vary. Some notes about doing this include:
 - Use the result of the `wavl_get_first()` routine as the *element* argument to `wavl_get_next()`.

- Use the result of `wavl_get_next()` as the `element` argument to subsequent calls to `wavl_get_next()`.
- Pass the same `handle` parameter each time to walk the entire tree.

Note Regardless of the method you use to walk the WAVL tree, you must choose a `thread` (AVL tree) to walk to determine the order in which the nodes are visited.

21.3.2.5 Search a Wrapped AVL Tree

To find a node on an AVL tree that is part of an AVL tree, use the `wavl_search()` routine. This function is illustrated in `/vob/ios/sys/atm/atm_arpserv.c` and has a syntax of:

```
wavl_node_t *wavl_search(wavl_handle_type *handle,
                         wavl_node_type *target,
                         int thread)
```

where:

- The `handle` parameter is a pointer to a previously allocated handle that has been initialized with `wavl_init()`.
- The `node` parameter points to a `wavl_node_type` member of a previously allocated structure.
- The `thread` parameter indicates which AVL tree of the WAVL tree is to be searched. This also indicates which comparison function specified to `wavl_init()` should be called to compare nodes in the AVL tree with the target node.
- A pointer to the node that compared equally to the `target` node is returned, or `NULL` if no such node is found.

In the example in `/vob/ios/sys/atm/atm_arpserv.c`, the `compare_ip_addr()` function is registered as the `compare function` for this AVL tree during the call to `wavl_init()`. This comparison function works as follows:

- 1 It treats the parameters (`n1` and `n2`) as `avl_node_type` pointers even though they are `wavl_node_type` pointers. This works because the first member of a `wavl_node_type` is an `avl_node_type`.
- 2 The comparison uses the `wavl_normalize()` function to return a pointer to the base of the array of `wavl_node_type` structures within a containing structure.
- 3 The `wavl_to_entry()` routine is then called with this address to return a pointer to the beginning of the containing structure (much like a `findblock` routine).
- 4 It compares the `ipaddr` members of the two containing structures and returns an appropriate result.

Note As an alternative to `wavl_search()`, you may call `avl_search()` on the appropriate AVL tree directly, passing the appropriate arguments including the comparison function. However, this defeats the purpose of using WAVL trees.

21.3.2.6 Remove a Node from a WAVL Tree

To delete nodes from a tree, use the `wavl_delete()` function. The syntax of this function is:

```
wavl_node_type *wavl_delete(wavl_handle *handle, wavl_node_type *node)
```

where:

- The `handle` parameter is a pointer to a previously allocated handle that has been initialized with `wavl_init()`.
- The `node` parameter points to a `wavl_node_type` member of a previous structure that was previously added to the tree. In the example above, this parameter is set to `our_ptr`, which was returned by a call to `wavl_search()`.
- The value returned is a pointer to the deleted node, or `NULL` if the node could not be deleted. Common causes for this include that `handle` or `node` are invalid. The example code does not check the return value because it does not care whether or not the deletion succeeds.

After the call to `wavl_delete()`, the space associated with the containing structure should be freed if it was allocated with `malloc()`. This can also be accomplished using a call to `free()`.

Note To remove a node from only one of the AVL trees associated with an WAVL tree, use `wavl_delete_thread()`.

21.3.2.7 Reset Pointers

To reset the pointers to the start of the tree structure, use the `wavl_normalize()` function. This function does not adjust any pointers.

```
static inline wavl_node_type * wavl_normalize(avl_node_type *node,  
int thread);
```

21.3.2.8 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);
```

21.3.3 Manipulate Threaded AVL Trees

The following information on Threaded AVL trees is from EDCS-258445. Please refer to that document for detailed information on Threaded AVL Trees.

Inorder traversal is a common operation in a binary search tree. To do this in an ordinary balanced tree (AVL or RB), we need to maintain a list of the nodes above the current node, or at least a list of the nodes still to be visited. This leads to the requirement of a stack either implicitly or explicitly.

A stack is not desired for two reasons. First, stacks take up space. Second, they are fragile; if an item is inserted into or deleted from the tree during traversal, or if the tree is balanced, we have to rebuild the traverser's stack.

This section is aimed at gaining fast sequential access in AVL trees by adding a special pointer called a “thread” to the right side of the nodes, producing what is called a “right-threaded AVL tree”. Since inorder traversal is a common operation in binary search trees, such a tree is right-threaded only.

Note The current IOS RB implementation does support threads, but the memory overhead is more when compared to that of a right-threaded AVL implementation.

21.3.3.1 Benefits of Right-Threaded Trees

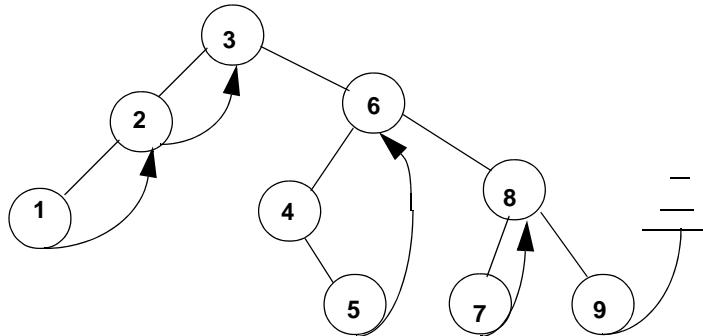
The existing AVL data structure is not robust across tree transformations. That is, say there are multiple walker, inserter, and deleter threads accessing the same AVL tree. At one point, the walker threads get suspended on a node and, before they resume, the inserters and deleters transform the tree. Now the walkers should proceed appropriately instead of simply returning. The appropriate procedure is incorporated into right-threaded trees. It might be helpful when the job of tree accessing / transforming has been split across multiple threads to minimize response time. Potential beneficiaries are ISDN, IPMULTICAST, MPLS, RSVP, IPROUTING, IGRP, CLNS, VOIP, ATM, DAILER, SGCP, LES, TBRIDGE, VLAN, and SNMP.

In an ordinary AVL tree, a lot of pointers go unused. These are used to store null pointers. In a right-threaded tree, a node's right child pointer field, if it would otherwise be a null pointer, is used to point to the node's inorder successor. The greatest valued node has a null pointer for its right thread.

21.3.3.2 An Example of a Right-Threaded Tree

Figure 21-1 is an example of the right-threaded tree.

Figure 21-1 Example of Right-Threaded Tree



In Figure 21-1, the circles represent nodes of an AVL right-threaded tree. The arrows represent pointers which now point to the node's inorder successor.. Node 9's pointer points to null.

The right-threaded tree data is protected across concurrent accesses by walker, deleter, and inserter threads. Before AVL walk blocks on the node (because of MORE or any other), it memlocks the node. So the content of the node is intact across deletes.

21.3.3.3 How Is the Threaded AVL Node Protected?

There are three ways in which a threaded AVL Node is protected:

- A node is blocked by the walker thread and the insert happens in the vicinity of that node.

The insert function will adjust the pointers in such a way that when the walk resumes, it continues with respect to the current nodes, latest inorder successor.

- A node is blocked by the walker thread and the delete happens in the vicinity of that node.
The delete will adjust the pointers in such a way that when the walk resumes, it continues with respect to the current nodes, latest inorder successor.
- A node is blocked by the walker thread and the delete happens on the same node.
There is a flag (AVL_FREE, no extra memory required) set in the deleted, unfreed node that tells the status of the node. When a delete function deletes the node from the tree it sets the status of the node to AVL_FREE. Now, when the walk resumes, the first thing it will do is to check the status of the node. If the node is found deleted, the tree is searched to find the minimum of the nodes with keys greater than the key of the current suspended node and resumed. The key in the suspended node is intact because of the memory lock done before the walker thread suspension.

This data structure is aimed at fast sequential access. In contrast to the existing AVL, the insert and delete operations are made non-recursive to have better performance. Also there shouldn't be any substantial increase in the memory per node of the tree.

The existing AVL APIs have been internally modified to use the right-threaded implementation.

21.3.3.4 Comparison with Other Trees

An optimized right-threaded AVL Tree Implementation seems to outweigh the existing AVL and RB Tree Implementations both in terms of time-complexity and space-complexity, as shown in Table 21-3.

Table 21-3 Space Complexity of Different Trees

Type of Tree	Constant Memory per Tree (in Bytes)	Memory Overhead per Node (in Bytes)
AVL Tree	0	12
Right-Threaded AVL Tree	0	10
RB Tree	136	56

A walk or get-next on right-threaded AVL trees is faster compared to one on existing (ordinary) AVL tree implementation. Also, insert and delete operations are faster due to the non-recursive implementation.

21.3.3.5 New AVL Node Structure

The new AVL node structure is

```
typedef struct avl_node_type_ {
    struct avl_node_type_ *avl_link[2];
    unsigned char avl_rtag;
    signed char avl_balance;
} avl_node_type;
```

`avl_link` is basically the left and right child pointers. `avl_rtag` says whether the right pointer is actually child or a pointer to the inorder successor. `avl_balance` is the balance factor [right ht - left ht]. (ht is the tree height).

These fields are transparent to the applications. None should use these.

21.3.3.6 New Threaded AVL Functions

The AVL library exports the following new API functions shown in Table 21-4:

Table 21-4 New Threaded AVL Functions

API Function	Description	Prototype
<code>avl_get_last()</code>	To get the greatest node in the tree or NULL if there is no tree.	<code>avl_node_type *avl_get_last (avl_node_type *top);</code>
<code>avl_get_next_threaded()</code>	To get the next element of the input parameter <code>element</code> .	<code>avl_node_type *avl_get_next_threaded (avl_node_type *element);</code>
<code>avl_is_leaf()</code>	To check whether a node is a leaf.	<code>boolean avl_is_leaf (avl_node_type *node);</code>
<code>avl_walk_extended()</code>	To walk the AVL tree in inorder, calling the walker function at every visited node.	<code>boolean avl_walk_extended (avl_node_type **top, avl_walker_type proc,void *paramptr, void (*lockfunc) (void *), void (*freefunc) (void *), avl_compare_type compare_func);</code>

21.3.3.7 How to Use Threaded AVL Functions

An AVL node is the basic structure around which the tree is built. To use these routines, you need to embed the new AVL structure node in the application structure at the very top.

For example:

```
typedef footype_ {
    avl_node_type avlnode;
    int frobnitz;
    widgettype widget;
} footype;
```

Next, you need to define a comparison routine which compares two footypes and returns an `avl_compare`

```
enum avl_compare foo_compare (footype node_one, footype node_two)...
```

You need to define a comparison routine to return comparisons about two elements in the tree. `AVL_LT` implies that `node_one < node_two`, `AVL_EQ` implies that `node_one == node_two`, and `AVL_GT` implies that `node_one > node_two`.

Finally, for the AVL functions that need to be used (`avl_insert()`, `avl_delete()`, `avl_walk()`, etc.), you need to define some static inline wrappers which simply cast arguments.

For example:

```
static inline boolean foo_get_next (footype *current, footype **next)
{
    return (avl_get_next ((avl_node_type *) current, (avl_node_type **) next,
        (avl_compare_type) foo_compare));
}
```

21.3.4 Manipulate AVLDup Trees

The following information on AVLDup trees is taken from ENG-163380 document. Please refer to that document for detailed information on AVLDup trees.

AVL trees are balanced search trees which are one of the types of binary search trees that are supported in IOS. One restriction of AVL trees is that you cannot have multiple elements with the same key value on the tree. There are some applications where this restriction can not be coped with. AVL Duplicate trees are being introduced in order to remove the restriction that AVL trees have, in which multiple elements with the same key value are not allowed on the tree.

21.3.4.1 Initialize an AVL Duplicate Tree

To initialize an AVL duplicate tree, perform the following steps. Allocate and initialize a user data element which is to become the first entry in the tree. Setup the root of the tree to be a pointer of type `avldup_node_type*` and pointing to `NULL`. Then pass this root as the `top` parameter and a pointer to the newly allocated user data element as the `new_data` parameter in the `avldup_insert()` function. This creates a newly created node as the “root” or topmost node, in the AVL tree.

21.3.4.2 Insert a Node into an AVLDup Tree

To insert an initialized node, which is *not* in the tree, into the tree, call the `avldup_insert()` function.

```
avldup_node_type *avldup_insert(avldup_node_type **top,
                                void *new_data,
                                boolean *balancing_needed,
                                avldup_compare_avldup_and_user_node_type
                                compare_avldup_and_user_node_func,
                                avldup_compare_avldup_nodes_type
                                compare_avldup_nodes_func,
                                avldup_user_data_list_compare_type
                                compare_avldup_user_data_list_func);
```

21.3.4.3 Delete a Node in an AVLDup tree

To delete the user data element, call the `avldup_delete()` function.

```
avldup_node_type *avldup_delete(avldup_node_type **top,
                                 void *data_to_delete,
                                 boolean *balancing_needed,
                                 avldup_compare_avldup_and_user_node_type
                                 compare_avldup_and_user_node_func,
                                 avldup_compare_avldup_nodes_type
                                 compare_avldup_nodes_func);
```

21.3.4.4 Search an AVLDup tree

To search the tree using the supplied comparison function, call the `avldup_search()` function.

```
avldup_node_type *avldup_search(avldup_node_type *top,
                                void *user_data_goal,
                                avldup_compare_avldup_and_user_node_type
                                compare_avldup_and_user_node_func);
```

21.3.4.5 Walk an AVLDup tree

To walk the AVLDup tree, calling the specified procedure at each node, call the `avldup_walk()` function.

```
boolean avldup_walk (avldup_node_type *element,
                     avldup_walker_type proc,
                     void *paramptr)
```

21.3.4.6 Retrieve Next AVLDup Node

To get the least node greater than the value passed in `element`, call the `avldup_get_next()` function.

```
avldup_node_type *avldup_get_next (avldup_node_type *top,
                                   avldup_node_type *element,
                                   avldup_compare_avldup_nodes_type compare_avldup_nodes_func)
```

21.3.4.7 Retrieve First AVLDup Node

To get the least node in an AVL Duplicate tree, call the `avldup_get_first()` function.

```
avldup_node_type *avldup_get_first (avldup_node_type *top)
```

21.4 Manipulate Radix Trees

21.4.1 Initialize a Radix Tree

To initialize a radix tree, use the `rn_inithead()` function.

```
int rn_inithead(void **head, int off)
```

21.4.2 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)
```

21.4.3 Traverse a Radix Tree

Table 21-5 lists the functions available for traversing (walking) a radix tree.

Table 21-5 Functions for Traversing a Radix Tree

Walking Action	Function
Walk a radix tree, calling a function for every node found in the tree.	<code>int rn_walktree(struct radix_node *rn, rn_walk_function function, ...)</code>
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.	<code>int rn_walktree_blocking(struct radix_node *rn, rn_walk_function function, ...)</code>

Table 21-5 Functions for Traversing a Radix Tree (continued)

Walking Action	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.	<code>int rn_walktree_blocking_list(struct radix_node *rn, rn_walk_function function, va_list pointer)</code>
Walk a radix tree, dismissing control of the processor in the middle of the walk to allow other threads to run.	<code>int rn_walktree_timed(struct radix_node_head *head, rn_walk_function walker, rn_succ_function nextnode, ...)</code>
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.	<code>int rn_walktree_version(struct radix_node *head, u_long version, rn_walk_function function, rn_succ_ver_function nextnode, ...)</code>

21.4.4 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);
```

21.4.5 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);
```

21.4.6 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);
```

21.5 String Database for Fast Lookup

21.5.1 Design Overview

The string database provides a method for subsystems to perform fast lookups on strings and retrieve a context which they registered with the string. The database uses a hash table for its lookup and the hash function is currently a 32 bit CRC of the string. The bucket collision logic first compares the CRC, then compares the length and finally performs a string comparison to find an exact match. Every string added to the string database is stored in the same hash table, thus amortizing the memory needed for the hash table across multiple subsystems. Binary strings (i.e. non-ASCII, non-null terminated) are supported. This allows IP addresses or MAC addresses to be used with the string-db. Additionally, ASCII strings are supported as well as case-sensitive/insensitive searches on ASCII strings.

21.5.2 API Overview

The following APIs are discussed:

- `sdb_register_component()`
- `sdb_add_string()`
- `sdb_remove_string()`
- `sdb_find_string()`
- `sdb_string_addr()`

21.5.2.1 `sdb_register_component()`

The very first thing clients of the string-db must do is to register with the string-db to obtain a unique *owner_id*. During this registration the client tells the string-db whether it wants to be able to perform case insensitive searches on strings it adds. It also tells the string database whether it is allowed to add more than one context to a string it has already added to the database. The client is returned a unique id which it must use in all subsequent calls to the string-db.

To register with the string database, call the `sdb_register_component()` function.

```
#include "../os/string_db.h"
boolean sdb_register_component(char *component_name,
                               sdb_handle (*get_sdb_handle)(void *),
                               boolean unique,
                               ushort *owner_id);
```

21.5.2.2 `sdb_add_string()`

Once the client has registered with the string database, it is then acceptable to add strings. Whenever the client wishes to add a string to the database, it must supply the string, its unique owner ID obtained through the registration process, the context to be associated with the string, and the type of string (binary, case sensitive, insensitive ASCII). The client is told whether the operation was successful and is returned an opaque handle for that string which MUST be stored and used in the remove call.

To add a string to the database and associate this owner with the string, call the [sdb_add_string\(\)](#) function.

```
#include "../os/string_db.h"
boolean sdb_add_string(char *buf,
                      int len,
                      sdb_string_types type,
                      sdb_handle *handle,
                      ushort owner_id,
                      void *owner_context);
```

21.5.2.3 sdb_remove_string()

When the client desires to remove the string, it should call this routine to do so.

To request from the owner to disassociate himself from the string, call the [sdb_remove_string\(\)](#) function.

```
#include "../os/string_db.h"
boolean sdb_remove_string(sdb_handle handle, void *owner_context)
```

21.5.2.4 sdb_find_string()

Find the a string and return the context associated with that string. In the case where more than one context is stored with that string, an index can be used to retrieve the nth context associated with the string.

To find a match on a string in the string database, call the [sdb_find_string\(\)](#) function.

```
#include "../os/string_db.h"
boolean sdb_find_string(char *buf,
                       int len,
                       sdb_string_types type,
                       int owner_index,
                       ushort owner_id,
                       void **owner_context);
```

21.5.2.5 sdb_string_addr()

To return a pointer to the string associated with the handle returned from the [sdb_add_string\(\)](#) function, call the [sdb_string_addr\(\)](#) function.

```
#include "../os/string_db.h"
char *sdb_string_addr(sdb_handle handle,
                      int *len);
```

Queues and Lists

22.1 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.

Note Cisco IOS queues and lists development questions can be directed to the interest-os@cisco.com and os-infra-team@cisco.com aliases.

22.1.1 Singly Linked Lists (Queues)

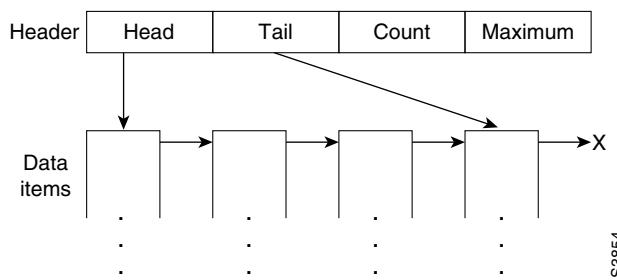
In the original version of the Cisco IOS software, the data structure for singly linked lists was a simple queue in which items were added at the end (tail) of the queue and removed from the beginning (head) of the queue. This simple data structure has developed into a singly linked list structure in 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.

Figure 22-1 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 22-1 Direct Queues

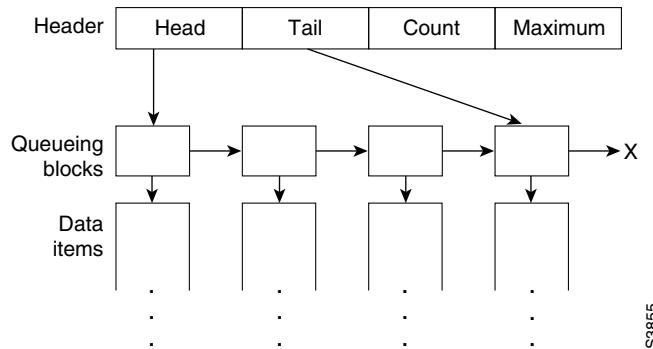


S3854

- 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.

Figure 22-2 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 22-2 Indirect Queues



22.1.2 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. For more information, see section 22.5, “Manipulate Simple Doubly Linked Lists.”
 - 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. For more information, see section 22.6, “Manipulate Doubly Linked Lists with the List Manager.”

22.2 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.

22.2.1 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.

Prior to 12.2T:

```
void queue_init(queuetype *queue, int maximum);
```

New in 12.2T:

```
boolean queue_init(queuetype *queue, int maximum);
```

22.2.2 Determine the State of a Queue

Table 22-1 describes the macros you can use to determine the state of a singly linked list.

Table 22-1 Macros for Determining the State of a Queue

Task	Function
Determine whether a queue is empty.	boolean QUEUEEMPTY(queuetype *queue)
Determine whether a queue is full.	boolean QUEUEFULL(queuetype *queue)
Determine whether a queue has a specified amount of space.	boolean QUEUEFULL_RESERVE(queuetype *queue, int reserve)
Determine the number of items on a queue.	int QUEUESIZE(queuetype *queue)

22.2.3 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);
```

22.3 Manipulate Direct Queues

22.3.1 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.

22.3.1.1 Add an Item to a Queue

maximum is specified in `queue_init()` and is ignored in the following functions.

Manipulate Direct Queues

To add an item to the end of a queue, use the `enqueue()` function.

Prior to 12.2T:

```
void enqueue(queuetype *queue, void *data);
```

New in 12.2T:

```
boolean enqueue(queuetype *queue, void *data);
```

To add an item to the beginning of a queue, use the `requeue()` function.

Prior to 12.2T:

```
void requeue(queuetype *queue, void *data);
```

New in 12.2T:

```
boolean requeue(queuetype *queue, void *data);
```

To insert an item at a relative position in a queue, use the `insqueue()` function.

Prior to 12.2T:

```
void insqueue(queuetype *queue, void *data, void *previous);
```

New in 12.2T:

```
boolean insqueue(queuetype *queue, void *data, void *previous);
```

22.3.1.2 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.

Prior to 12.2T:

```
void unqueue(queuetype *queue, void *data);
```

New in 12.2T:

```
boolean unqueue(queuetype *queue, void *data);
```

22.3.1.3 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);
```

22.3.2 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.

22.3.2.1 Add an Item to a Queue

To add an item to the end of a queue, use the `p_enqueue()` function.

```
boolean p_enqueue(queueType *queue, void *data);
```

To add an item to the beginning of a queue, use the `p_requeue()` function.

```
boolean p_requeue(queueType *queue, void *data);
```

22.3.2.2 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(queue *queue);
```

To remove an item from an arbitrary point in a queue, use the `pak_unqueue()` function.

```
boolean p_unqueue(queue *queue, void *data);
```

To remove the next item after an arbitrary point in a queue, use the `p_unqueueenext()` function.

```
boolean p_unqueueenext(queue *queue, void **previous);
```

22.3.2.3 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);
```

22.4 Manipulate Indirect Queues

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.

22.4.1 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(queue *queue, paktype *pak);
```

```
void data_enqueue(queue *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(queue *queue, paktype *pak, elementtype *previous);
```

To add an item at any arbitrary position in an indirect queue, use the `data_insertlist()` function. This function provides 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);
```

22.4.2 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);
```

22.4.3 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);
```

22.4.4 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);
```

22.4.5 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);
```

22.5 Manipulate Simple Doubly Linked Lists

The doubly linked list functions provide a fast, straightforward method to build a circular doubly linked list. These functions manipulate lists of `dqueue_t` structures. The functions are:

```
o_init()
lw_insert()
lw_remove()

o_enqueue()
o_dequeue()
o_unqueue()
```

For most purposes, you should use `lw_insert()` and `lw_remove()`, both of which ignore the embedded timestamp in the `dqueue_t` structure, and possibly `o_init()` to initialize the doubly linked list.

The functions `o_enqueue()`, `o_dequeue()`, and `o_unqueue()` are too specialized for general-purpose use because they pay attention to a timer structure that is embedded into the `dqueue_t` structure. In particular, `o_enqueue()` inserts entries in the list in order of timer expiration times and `o_dequeue()` will not dequeue an element until its timer has expired.

The `dqueue_t` structure is used to build these lists:

```
typedef struct dqueue_ {
    struct dqueue_ *flink;
    struct dqueue_ *blink;
    void           *parent;
    sys_timestamp   value;
} 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.

22.5.1 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);
```

22.5.2 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);
```

22.5.3 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);
```

22.6 Manipulate Doubly Linked Lists with the List Manager

22.6.1 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.

See also the following macros:

- [FOR_ALL_ELEMENTS_IN_LIST](#)
- [FOR_ALL_DATA_IN_LIST](#)
- [FOR_ALL_DATA_IN_LIST_FROM_ELEMENT](#)
- [LIST_GET_DATA](#)
- [LIST_SET_DATA](#)
- [LIST_VALID](#)
- [LIST_SIZE](#)
- [LIST_FULL](#)
- [LIST_EMPTY](#)
- [LIST_NEXT_ELEMENT](#)
- [LIST_PREV_ELEMENT](#)
- [LIST_HEAD_ELEMENT](#)
- [LIST_TAIL_ELEMENT](#)
- [LIST_ELEMENT_QUEUED](#)
- [ELEMENT_GET_LIST](#)

22.6.2 Create a List

To create a new list, use the `list_create()` function.

```
list_header *list_create(list_header *list, ushort maximum, char *const name,  
                        ushort flags)
```

22.6.3 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_FLAGS_AUTOMATIC` controls whether the list manager should create and delete `list_element` data structures automatically.
- `LIST_FLAGS_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);
```

22.6.4 Add an Item to a List

To add an item to the end of a list, use the `vlist_enqueue()` function.

```
static inline void *list_enqueue(list_header *list, list_element *element,
void *data);
```

To add an item to the middle of a list, use the `list_insert()` function.

```
static inline void *list_insert(list_header *list, list_element *element,
void *data,
list_insert_func_t func);
```

To add an item to the beginning of a list, use the `list_requeue()` function.

```
static inline void *list_requeue(list_header *list, list_element *element,
void *data);
```

22.6.5 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);
```

22.6.6 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.

```
static inline void *list_remove(list_header *list, list_element *element,
void *data);
```

22.6.7 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.

Table 22-2 lists the default behaviors for these wrappers.

Table 22-2 List Action Vector Default Behavior

Function	Default Behavior
<code>list_dequeue()</code>	Remove an item from the beginning of the list.
<code>list_enqueue()</code>	Add an item to the end of the list.
<code>list_insert()</code>	Use the provided function to determine where to add the new item.
<code>list_remove()</code>	Remove the specified item.
<code>list_requeue()</code>	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);
```

22.6.8 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);
```

22.6.9 Display the Contents of a List

To specify a display function to display a list item, for use when showing 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);
```

22.6.10 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);
```

22.6.11 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_create(&procq_ready_h, 0, "Sched High", LIST_FLAGS_INTERRUPT_SAFE);
list_create(&procq_ready_m, 0, "Sched Normal", LIST_FLAGS_INTERRUPT_SAFE);
list_create(&procq_ready_l, 0, "Sched Low", LIST_FLAGS_INTERRUPT_SAFE);
list_create(&procq_idle, 0, "Sched Idle", LIST_FLAGS_INTERRUPT_SAFE);
list_create(&procq_dead, 0, "Sched Dead", LIST_FLAGS_INTERRUPT_SAFE);
```

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);
```

22.7 Other Doubly Linked List Functions

There are other functions which manipulate doubly linked functions. The `dllobj_t` structure is used to build these lists:

```
typedef struct dllobj {
    int count;
    dllobj_links *head;
    dllobj_links *tail;
} dllobj_t
```

22.7.1 Add Element to End of List

To add the user specified object `element` to the end of the list, call the `dllobj_add()` function.

```
#include "sys/util/dllobj.h"
void dllobj_add (dllobj_t *list, void *element);
```

22.7.2 Add Element After Specified Object

To add the user specified object `element` after the entry identified by `after`, call the `dllobj_add_after()` function.

```
#include "sys/util/dllobj.h"
void dllobj_add_after (dllobj_t *list, void *after, void *element);
```

22.7.3 Add Element Before Specified Object

To add the user specified object *element* BEFORE the entry identified by *before*, call the `dllobj_add_before()` function.

```
#include "sys/util/dllobj.h"
void dllobj_add_before (dllobj_t *list, void *before, void *element);
```

22.7.4 Add Element to Head of List

To add the user specified object *element* to the HEAD of the list, call the `dllobj_add_to_front()` function.

```
#include "sys/util/dllobj.h"
void dllobj_add_to_front (dllobj_t *list, void *element);
```

22.7.5 Search For Element in List

To confirm or deny depending if the element *elem* is in the specified list *list*, call the `dllobj_in_list()` function.

```
#include "sys/util/dllobj.h"
boolean dllobj_in_list (dllobj_t *list, void *elem);
```

22.7.6 Initialize Linked List Management Object

To initialize a linked list management object, call the `dllobj_init()` function.

```
#include "sys/util/dllobj.h"
void dllobj_init (dllobj_t *list);
```

22.7.7 Insert Element In The Ordering Specified

To insert into the list, depending on the ordering specified by the user specified function, call the `dllobj_insert_ordered()` function.

```
#include "sys/util/dllobj.h"
boolean dllobj_insert_ordered (dllobj_t *list,
                               comparer cmp_func,
                               void *elem,
                               boolean allow_duplicates);
```

22.7.8 Read Nth Element

To read the n'th element from the specified list, just like it was an array, call the `dllobj_nth()` function.

```
#include "sys/util/dllobj.h"
void *dllobj_nth (dllobj_t *list, int n);
```

22.7.9 Read First Element

To “pop” out the first entry from the linked list, call the `dllobj_read()` function.

```
#include "sys/util/dllobj.h"
void *dllobj_read (dllobj_t *list);
```

22.7.10 Remove Element From List

To remove the user specified object `element` from the `list`, call the `dllobj_remove()` function.

```
#include "sys/util/dllobj.h"
void dllobj_remove (dllobj_t *list, void *element);
```

22.8 Iterate a List or Queue Shared with Interrupt Level Code

If a CLI accesses a list or queue that is also used at interrupt level, a “shadow” structure is usually untenable. Therefore you must check the list or queue to make sure that it is still intact during your display by performing the following steps:

Step 1 Populate an `idblist_iterator_desc` structure with pointers to functions that can access the first, previous, and next entries in your list. For example, in `if/if_list.c`:

```
static iterator_desc_t idblist_iterator_desc = {
    idblist_iterator_first_entry,
    idblist_iterator_prev_entry,
    idblist_iterator_next_entry,
};
```

Note For singly linked lists, the function to obtain the previous entry in the list might not be very efficient since it would have to run through the list from the beginning.

Step 2 Create a unique iterator for the list or queue and get the first entry:

```
iterator = iterator_create(...)
iterated_entry = iterator_get_first_entry(...)
```

Step 3 Loop until the end of the list or queue or until an entry is deleted:

```
iterated_entry = iterator_get_next_entry(...)
if (iterated_entry == NULL)
break;
do-displays-for-this-entry(...)
```

Step 4 Upon completion of the iteration, delete the iterator:

```
iterator_delete(...)
```

22.9 Indexed Object Lists

Indexed objects are basically like arrays (not arrays since they are actually malloc’ed based on a size specification). They are good for a one-time insertion and multiple searches since their search complexity is O(log n). If elements have to be inserted, this would/could require shifting subsequent elements, as in the case of arrays.

22.9.1 Index Objects and the Comparison Function

The only thing the user has to do is to initialize an indexed object list and supply a “comparison function” during the initialization. This is the critical part of the object. The user supplied comparison function should be implemented as efficiently as possible and have *no* side effects (in

other words, should *not* change the indexed object list in any way). The index object then applies the user supplied comparison function, by passing the searched object *key* and the stored element pointers, in a binary search to locate the searched object itself. The comparison function should return 3 types of outputs, 0 for equality, < 0 for the first argument being less than the second argument and > 0 otherwise.

Note The comparison routine should be written to accept the *key* parameter *first* followed by the “user supplied” element pointer *second*. Note that the index object stores an element with the same key *only* once. In other words, duplicate objects are not allowed.

22.9.1.1 Syntax of the Comparison Function and Index Object

The following code shows the syntax of the comparison function and index object.

```
/*
 * syntax of comparison function
 */
typedef int (*indxobj_cmp_func) (void *key, void *elem);

/* the index object */
typedef struct indxobj {

    /* number of elements in index */
    int n;

    /* maximum capacity */
    int max;

    /* fixed size if set */
    int fixed_size;

    /* will be searched in interrupts ? */
    boolean searched_in_interrupts;

    /* user data pointers */
    void **elements;

    /* user supplied comparison function */
    indxobj_cmp_func compare_function;
} indxobj_t;
```

22.9.2 Indexed Object APIs

There are a number of APIs for manipulating indexed object lists. They are described in the following sections.

22.9.2.1 Initializing an Index Object

To initialize an index object specified by *ind*, call the [indxobj_init\(\)](#) function.

```
boolean indxobj_init (indxobj_t *ind, indxobj_cmp_func cmp, int size, boolean
searched_in_interrupts);
```

22.9.2.2 Adding an Element to the Indexed Object List

To add an element identified by *key* and specified by *user_data*, to the index object, call the [idxobj_add\(\)](#) function.

```
void *idxobj_add (idxobj_t *ind, void *key, void *user_data);
```

22.9.2.3 Removing an Element From an Indexed Object List

To remove the entry represented by the key *key* from the index, call the [idxobj_remove\(\)](#) function.

```
void *idxobj_remove (idxobj_t *ind, void *key);
```

22.9.2.4 Finding the Size of the Indexed Object List

To return the number of elements in the indexed object list, at any time, call the [idxobj_size\(\)](#) function.

```
int idxobj_size (idxobj_t *ind);
```

22.9.2.5 Searching for an Element in an Indexed Object List

There are several functions that will search for an element in an indexed object list.

To search and return a pointer to the user element identified by *key*, call the [idxobj_search\(\)](#) function.

```
void *idxobj_search (idxobj_t *ind, void *key, idxobj_cmp_func  
cmp_override);
```

To search and return a pointer to the user element immediately after the given *key*, call the [idxobj_search_next\(\)](#) function.

```
void *idxobj_search_next (idxobj_t *ind, void *key);
```

To search and return the location of a user element identified by *key* (in the indexed object list), call the [idxobj_find_position\(\)](#) function.

```
int idxobj_find_position (idxobj_t *ind, void *key, int *insertion_point,  
idxobj_cmp_func cmp_override);
```

To perform a linear search through all the elements, starting from a user specified point, call the [idxobj_linear_search\(\)](#) function.

```
int idxobj_linear_search (idxobj_t *ind, int start, void *key,  
idxobj_cmp_func cmp_function, void *user_ptr)
```

22.9.2.6 Returning a Pointer to the User Data at the Nth Value in the Indexed Object List

To return a pointer to the user data at the nth location in the indexed object list, call the [idxobj_get_nth_element\(\)](#) function.

```
void *idxobj_get_nth_element (idxobj_t *ind, int n);
```

22.9.2.7 Cleaning Up Indexed Object Lists

To reduce the element count of an index object to 0, clearing all the list, call the `idxobj_cleanup()` function.

```
void idxobj_cleanup (idxobj_t *ind, boolean free_storage);
```

22.9.3 Example:Indexed Object Lists

The example below illustrates the use of indexed object lists for ATM address management. Here is a detailed look at the example. The data to be manipulated is represented by the `atm_address_t` structure (defined below):

```
typedef struct atm_address {

    uchar address [STATIONLEN_ATMNSAP];
    uint refcount;
    ushort id;

} atm_address_t;
```

Note There are two indexed object lists considered here for ATM address management - one sorted on the 20-byte ATM addresses (`atm_address_index`), and the other on *id* (`atm_address_id_index`).

The first function that one needs to look at is the `idxobj_init()` function—the initialization routine for indexed object list. The indexed object list is initialized during the initialization of the ATM subsystem, via a call to the `idxobj_init()` function:

```
idxobj_init(&atm_address_index,
atm_addresses_compare, 0, FALSE);
    idxobj_init(&atm_address_id_index,
cmp_atmaddrs_by_id, 0, FALSE);
```

`atm_addresses_compare()` and `cmp_atmaddrs_by_id()` are the comparison routines used during search, insertion and removal from the indexed object lists.

The addresses are added into the indexed object list in the `add_atm_address()` function, via a call to the `idxobj_add()` function.

```
/* add it to the indexes */
    idxobj_add(&atm_address_index, address, atm_addr);
    idxobj_add(&atm_address_id_index, (void*) ((int) atm_addr->id),
atm_addr);
```

In the first case `address` is passed as the key, while in the second case the `id` is the key. In both cases the `atm_addr` represents the data added to the indexed object list.

The `find_atm_address_by_id()` and `find_atm_address_by_addr()` functions are used to look for the data in the indexed object lists (based on the `id` and the 20-byte address respectively). These function use the `idxobj_search()` to look for the data.

```
idxobj_search(&atm_address_id_index,
(void*) ((int) id), NULL);

idxobj_search(&atm_address_index, addr, NULL);
```

Note that in both cases the `cmp_override` function (the third parameter) is passed as `NULL`, indicating that `cmp_atmaddrs_by_id()` and `atm_addresses_compare()` should be used as the comparison functions for the search.

Finally, the addresses are removed from the indexed object lists in the `free_atm_address()` function, via a call the `idxobj_remove()` function.

```
idxobj_remove(&atm_address_index, atm_addr->address);
idxobj_remove(&atm_address_id_index, (void*) ((int) atm_addr->id));

static idxobj_t atm_address_index;
static idxobj_t atm_address_id_index;

/*
 * comparing atm addresses for THIS index object ONLY.
 */
static int atm_addresses_compare (void *key, void *elem)
{
    return cmp_atm_addresses_inline((uchar*) key,
                                    (uchar*) ((atm_address_t*) elem)->address);
}

/*
 * comparing atm address id's for THIS index object ONLY.
 */
static int cmp_atmaddrs_by_id (void *key, void *elem)
{ return ((int) key - (int) ((atm_address_t*)elem)->id); }

/*
 * initialize this system
 */
static void init_atm_addresses (void)
{
    idxobj_init(&atm_address_index, atm_addresses_compare, 0, FALSE);
    idxobj_init(&atm_address_id_index, cmp_atmaddrs_by_id, 0, FALSE);
    id_counter = 0;
    wrapped_around = FALSE;
}

/*
 * find an atm_address_t structure by its id
 */
atm_address_t *find_atm_address_by_id (ushort id)
{
    return idxobj_search(&atm_address_id_index, (void*) ((int) id), NULL);
}
/*
 * find an atm_address_t structure by its 20 byte address
 */
atm_address_t *find_atm_address_by_addr (uchar *addr)

{
    return idxobj_search(&atm_address_index, addr, NULL);
}
/*
 * just blindly add an atm address.  If it
 * already matches one, it returns that one
 * and increments the ref count.  Otherwise
 * just creates a new one.
*/
```

```

/*
atm_address_t *add_atm_address (uchar *address)
{
    atm_address_t *atm_addr;

    /* is it already there ? */
    atm_addr = find_atm_address_by_addr(address);
    if (atm_addr) {
        atm_addr->refcount++;
        return atm_addr;
    }

    /* create a new one */
    if (atm_address_chunks == NULL) {
        atm_address_chunks = chunk_create(sizeof(atm_address_t),
            32, (CHUNK_FLAGS_DYNAMIC | CHUNK_FLAGS_BIGHEADER), NULL, 0,
            "MPX atm address chunks");
    if (atm_address_chunks == NULL) return NULL;
    }
    atm_addr = chunk_malloc(atm_address_chunks);
    if (!atm_addr) return NULL;

    /* now get an unused id for it */
    if (!get_new_atmaddr_id_index(&atm_addr->id)) {
        chunk_free(atm_address_chunks, atm_addr);
        return NULL;
    }

    /* initialize its fields */
    atm_addr->refcount = 1;
    bcopy(address, atm_addr->address, STATIONLEN_ATMNSAP);

    /* add it to the indexes */
    idxobj_add(&atm_address_index, address, atm_addr);
    idxobj_add(&atm_address_id_index, (void*) ((int) atm_addr->id),
               atm_addr);
    return atm_addr;
}

/*
 * Un-reference an atm address object directly.
 * If refcount drops to 0, just get rid of it.
 */
void free_atm_address (atm_address_t *atm_addr)
{
    if (!atm_addr) return;
    atm_addr->refcount--;
    if (atm_addr->refcount <= 0) {
        idxobj_remove(&atm_address_index, atm_addr->address);
        idxobj_remove(&atm_address_id_index, (void*) ((int) atm_addr->id));
        chunk_free(atm_address_chunks, atm_addr);
    }
}

```

22.10 Index Tables

(Index Table APIs are new in 12.2T and also available in 12.0S)

The following are the functions for a generic index table that uses numeric indexes. References to any type of structure can be stored in the table. The API is intended to be platform-independent. The size of the table is not bound to a constant value; i.e., the table grows as it is populated. The APIs can be called directly from any subsystem that wants to use index tables.

The following definition provides a handle to an index table, while keeping it opaque. The constructor `indxtbl_create()` should be called to instantiate the table.

```
typedef struct index_table_ *index_table_ptr_t;
```

22.10.1 Creating an Index Table

To create an index table, call the `indxtbl_create()` function.

```
#include "util/index_table.h"
boolean indxtbl_create(index_table_ptr_t *handle);
```

22.10.2 Removing an Element from an Index Table

To remove a leaf entry in an index table, call the `indxtbl_delete_element()` function.

```
#include "util/index_table.h"
boolean indxtbl_delete_element(index_table_ptr_t tbl, ulong index);
```

22.10.3 Freeing all Nodes of an Index Table

To free all the tree nodes of an index table, call the `indxtbl_free()` function.

```
#include "util/index_table.h"
boolean indxtbl_free(index_table_ptr_t *handle);
```

22.10.4 Finding the First Element of an Index Table

To find the first element stored in a particular index table, call the `indxtbl_get_first_element()` function.

```
#include "util/index_table.h"
void *indxtbl_get_first_element(index_table_ptr_t tbl);
```

22.10.5 Finding the First Empty Cell of an Index Table

To find the index of the first empty cell in a particular index table, call the `indxtbl_get_first_vacancy()` function.

```
#include "util/index_table.h"
boolean indxtbl_get_first_vacancy(index_table_ptr_t tbl, ulong *vacancy);
```

22.10.6 Finding the Last Element Stored in an Index Table

To find the last element stored in a particular table, call the `indxtbl_get_last_element()` function.

```
#include "util/index_table.h"
void *indxtbl_get_last_element(index_table_ptr_t tbl);
```

22.10.7 Finding the Following Element in an Index Table

To find the next element stored in a particular index table following the element at index `after`, call `indxtbl_get_next_element()` function.

```
#include "util/index_table.h"
void *indxtbl_get_next_element(index_table_ptr_t tbl, ulong after);
```

22.10.8 Finding the Next Empty Cell in an Index Table

To find the index of the next empty cell in a particular index table following `after`, call the `indxtbl_get_next_vacancy()` function.

```
#include "util/index_table.h"
boolean indxtbl_get_next_vacancy(index_table_ptr_t tbl, ulong after,
                                 ulong *vacancy);
```

22.10.9 Getting Nth Element in an Index Table

To get the pointer to the element stored at a given index, call the `indxtbl_get_nth_element()` function.

```
#include "util/index_table.h"
void *indxtbl_get_nth_element(index_table_ptr_t tbl, ulong n);
```

22.10.10 Finding the Previous Element in an Index Table

To find the next element stored in the table preceding the element at index `before`, call the `indxtbl_get_prev_element()` function.

```
#include "util/index_table.h"
void *indxtbl_get_prev_element(index_table_ptr_t tbl, ulong before);
```

22.10.11 Inserting an Element into an Index Table

To insert an element into the index table, call the `indxtbl_insert_element()` function.

```
#include "util/index_table.h"
boolean indxtbl_insert_element(index_table_ptr_t tbl, void *element,
                               ulong index);
```

Switching

Several mechanisms are used in the Cisco IOS software to facilitate the forwarding of traffic with minimal delay. The switched path that a packet takes through a router is often dependent upon the Layer-3 protocol being routed or the hardware platform being used, and the path could depend upon the version of IOS software being run. As such, it can be difficult to see how a switching mechanism is working or what the implications would be of enabling a particular switching path in a given network topology. This chapter attempts to dispel some confusion by identifying all the switching paths in use on the various platforms at the time of this writing, April 2000. The first section is an overview. The others are more nitty gritty programming instructions.

- Section 23.1, “Switching, an Overview,” provides an architecture overview of each type of path.
- Section 23.2, “Writing Fast Switching Code,” is a detailed but dated (circa 1997) description of fast switching.
- Section 23.3, “Adding a CEF Feature,” provides explanation, guidelines, and sample code.
- Section 23.4, “FIB Subblocks,” describes how to create, manage, use, and debug a subblock for your CEF feature.

As of 12.3T, infrastructure was added in IOS to provide support for the next generation AAA (triple A—authentication, authorization, and accounting), SSS (Subscriber Service Switch), and SSG (Service Selection Gateway):

- Section 23.5, “Hardware Session and L2 Hardware Switching,” describes the API developed for locally terminated tunnels and layer 2 to layer 2 point-to-point connections.

As of 12.2S RLS7, infrastructure was added that plugs directly into the IOS L2HW API to provide additional support for the next generation AAA, SSS, and SSG:

- Section 23.6, “L2VPN Platform API,” describes the API needed to set up L2VPN (Layer 2 VPN (Virtual Private Network)) services in hardware.

Note Cisco IOS switching development questions can be directed to the ios-switch-coders@cisco.com alias.

Terms

With a few exceptions, the following definitions are from the glossary of *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.

1000-12000 router

(1000-12000 line) The x-thousand product line consists of hardware that is tailored to specific markets: the 12000 is for optical backbones and high performance; the 7000 is for enterprise backbones or the Internet edge; the 4000 is for concentrators and filtering on the edge of the corporate backbone; the 3000 and 2000 are for regional and remote sales offices; and the 1000 is for telecommuters and single-person offices.

access control list

(ACL) List of packet filtering rules to provide security features. Standard ACLs only provide filtering based on source IP address adjacency.

access list

List kept by routers to control entry or exit from the router for a number of services, for example to prevent packets with a certain IP address from leaving on a particular interface.

accounting management

Subsystems that are responsible for collecting network data about resource usage. (One of five categories of network management defined by ISO for the management of OSI networks.)

adjacency

Two nodes are said to be adjacent if they can reach each other via a single hop across a link layer. The adjacency database is a table of adjacent nodes, each entry holding the Layer-2 MAC-rewrite information necessary to reach the adjacent node. The entries in an adjacency database are called *adjacencies*. See also CEF and FIB.

autonomous switching

Feature on Cisco routers that provides faster packet processing by allowing the ciscoBus to switch packets independently, without interrupting the system processor.

card

A module that is inserted into an IOS system that provides network interface support and, in one case, route processing.

CEF

Cisco Express Forwarding. Switching mechanism with performance on a par with fast-switching but that also scales to support internet backbone requirements. CEF uses a forwarding information base (FIB) and an adjacency table. *See also* FIB, adjacency, and dCEF.

CxBus

(Cisco Extended Bus) Data bus for interface processors (*see*) on Cisco 7000 series routers; operates at 533 megabits per second (Mbps).

dCEF

Distributed CEF. To improve the scalability of high-end routers, the CEF tables are distributed to special intelligent line cards, such as VIP line cards (7500) or Gigabit Switch Router (GSR) line cards. See also CEF, FIB, and adjacency.

distributed environment

Environment in which slave processors perform interrupt-level route processing under the direction of a central route processor (RP). The slave processors are on line cards. Examples of distributed environments: the 7500, 12000, and 10000 series. Compare with nondistributed environment.

DMA

Direct memory access. The transfer of data from a peripheral device, such as a network interface card, into memory without that data passing through the microprocessor. DMA transfers data into memory at high speeds with no processor overhead. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code.*)

fast switching

Cisco feature in which a route cache is used to expedite packet switching through a router. If fast switching is enabled, the driver code will transfer control temporarily to the fast switching code, which searches the route cache for a frame and other information constructed from a previously transmitted packet. If the route cache contains an entry, the fast switching code will attempt to send the packet directly to the destination interface. If the interface is busy, the packet is placed on the queue for that interface. There are several types of platform-specific fast switching techniques. *Compare with process switching.*

FIB

Forwarding information base (common ISO usage). Database of information used to make forwarding decisions. It is conceptually similar to a routing table or route-cache but very different in implementation. See also CEF and adjacency.

frame header

One of the headers in a data buffer, such as those specified in IEEE 802.3 and 802.5.

Interface Processor

(IP) Before VIPs came into being, all the different 7000 and 7500 cards were known as “IPs.” In fact, VIP stands for “Versatile Interface Processor.” For example, the ATM Interface Processor (AIP) is the ATM network interface for the 7000 series, designed to minimize performance bottlenecks at the UNI (User-Network Interface, an ATM Forum term that goes with NNI, Network-to-Network interface, and SNI, Subscriber-Network Interface). Other IPs include the Channel Interface Processor (CIP), Ethernet Interface Processor (EIP), Fast Ethernet Interface Processor (FEIP), FDDI Interface Processor (FIP), Fast Serial Interface Processor (FSIP), HSSI Interface Processor (HIP), MultiChannel Interface Processor (MIP), Serial Interface Processor (SIP), and Token Ring Interface Processor (TRIP).

line card

(LC) “LC” implies some autonomous intelligence and independent processing, as with the VIP cards for the 7500 series and, later, all of the GSR line cards. When an LC has a CPU onboard, it does route processing with the assistance of CEF chip-sets built into the board. LCs are wide cards, able to handle many interfaces and having multiple physical interface types.

load balancing

In routing, the ability of a router to distribute traffic over all its network ports that are the same distance from the destination address. Good load-balancing algorithms use both line speed and reliability information. Load balancing increases the utilization of network segments, thus increasing effective network bandwidth.

MTU

Maximum transmission unit. Maximum packet size, in bytes, that a particular interface can handle.

NDB

Network descriptor block. Mechanism used by a routing information base (RIB) to distribute the best route to a prefix to all of its clients. (From the RIB perspective, CEF is a redistribution client.) *See also RDB.*

network controller

1. CPU on board the network interface module that carries out status and control functions and performs intelligent operations
2. Component of the network interface that communicates with the core CPU.

network header

In a data buffer, the protocol datagram header; for example, IP, AppleTalk, or IPX headers. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.)

network interface

1. Boundary between a carrier network and a privately-owned installation
2. Unit (card) that plugs into a router slot and is used for the sending and receiving of packetized data. Also called a network interface module (NIM.)
3. Network protocol, for example Ethernet, Fast Ethernet, Token Ring, Serial, ISDN-BRI, ISDN-PRI, T1, E1, HSSI, IBM channel, ATM, FDDI
4. Hardware component of the unit that implements the physical layer. Also called the *port*.

Network Interface Module

(NIM) A non-hot-swappable card that is installed in the c4000 and c4500 series. It is relatively “low tech” in that it can handle only one physical layer type per NIM, for example four Ethernet ports and two Token Ring. NIMs sometimes have hardware assist but it is limited to physical layer framing. The NIM may be considered a forerunner to the PA, except that it is only usable on the c4000/c4500 series and the NIMs themselves do not usually contain any intelligence.

While the term “NIM” is not used to describe any of the cards inserted into the 7000-7500 series, a smaller entity used by the older 7500 IPs had a similar name, “network module.” These were for configuring different interface types, for example the FSIP had one that supported four ports, so that you could have a 4-port FSIP, or an 8-port FSIP if you installed two. These are not to be confused with “network interface modules (NIMs).”

nondistributed environment

Environment in which a single route processor (RP) performs all routing tasks (without the help of slave processors.) *Compare with distributed environment*.

particle

Composed of a data block and a particletype structure (used for managing the particle’s data block). A particle-based buffer is assembled from several particles, which are evenly-sized blocks that are smaller than the maximum MTU-size packet of an interface. A packet may not fit into a single particle and therefore would be “scattered” in blocks located randomly in memory.

plug-in

Thankfully, within the IOS software world, we have a generic term. A “plug-in” is anything that plugs in, regardless of whether it’s a line card, port adaptor, port module, or other type of card.

Port Adaptor

(PA) Unlike the GSR LCs, VIP LCs have no interfaces integrated into them, so PAs need to be added. PAs are widely used interface cards with a PCI bus connector. Two can be installed per VIP card and the same ones fit into the 7200 platform. PAs can be found in several other places besides the 7200 and VIP, such as the Cat6K FlexWan line card. Although some PAs have intelligence, none do distributed switching like the VIP LC. On the 7200 platform, PAs can be hot-swapped, but not on the VIP. The VIP itself is hot-swappable.

Port Module

(PM) The PM is basically a repackaged PA that is used in the 3600 and 2600 series. Electrically, it is exactly the same as a PA. However, for marketing reasons, the metalwork is different: you cannot take a PM and put it in a PA slot. The other difference is that PMs are not designed to be hot-swappable.

punting

(Ciscoism) Action by a device driver of sending a packet “up” to the next slowest switching level when the attempt at a lower level has not yielded a path or the packet could not be switched at that level for another reason. In the case of CEF, the levels are as follows, from lowest to highest: distributed CEF (dCEF), CEF, fast switching, route processing. For more information on the role of drivers, see “Cisco IOS Architecture” in *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.

queue

Central mechanism of the Cisco IOS software. The paktype structure contains a pointer to another paktype structure. These pointers may be chained together to form software-based packet queues, making it possible to move a packet from one queue to another by pointer manipulation, rather than copying or moving the data buffer.

RDB

Route descriptor block. Used by the routing information base (RIB) to pass path information about a given prefix. RDBs are components of the network descriptor block. The NDB describes the route whereas the RDB describes the paths available to the route.

receive ring

Used by hardware devices to buffer packets received on an interface. Each ring entry has a descriptor that contains pointers to the packet buffer, and the packet size and status of the entry, including ownership and error information. Receive descriptors are thought of as connected in a ring. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.) See also transmit ring.

route processor

(RP) 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*.

route/switch processor

(RSP) Processor module used in the Cisco 7500 series routers that integrates the functions of the route processor (RP) and switch processor (SP).

silicon switching

Switching based on the silicon switching engine (SSE), which allows the processing of packets independent of the silicon switch processor (SSP) system processor (*see*). Silicon switching provides high-speed, dedicated packet switching.

switch processor

(SP) Cisco 7000 series processor module that acts as the administrator for all CxBus activities. Sometimes called *ciscoBus controller*.

silicon switching engine

(SSE) Routing and switching mechanism that compares the data link or network layer header of an incoming packet to a silicon-switching cache, determines the appropriate action (routing or bridging), and forwards the packet to the proper interface. The SSE is directly encoded in the hardware of the SSP (silicon switch processor) of a Cisco 7000 series router. It can therefore

perform switching independently of the system processor, making the execution of routing decisions much quicker than if they were encoded in software. *See also* silicon switching and silicon switch processor.

silicon switch processor

(SSP) High performance silicon switch for Cisco 7000 series routers that provides distributed processing and control for interface processors. The SSP leverages the high-speed switching and routing capabilities of the silicon switching engine (SSE) to dramatically increase aggregate router performance, minimizing performance bottlenecks at the interface points between the router and a high-speed backbone. *See also* silicon switching and silicon switching engine.

transmit ring

Used by hardware devices to buffer packets to be transmitted over an interface. Each ring entry has a descriptor that contains pointers to the packet buffer, packet size, and status of the entry, including ownership and error information. Transmit descriptors are thought of as connected in a ring. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.) *See also* receive ring.

Unicast RPF

(unicast Reverse Path Forwarding) The goal is to verify if the source IP is reachable for the purpose of preventing malformed or forged source IP addresses from entering a network.

VIP

1. Versatile Interface Processor. Interface card used in 7000 and 7500 series routers. Provides multilayer switching and runs the Cisco IOS software. 2. virtual IP. Enables the creation of logically separated switched IP workgroups across the switch ports of a Catalyst 5000 running Virtual Networking Services software (on some Catalyst 5000 switches, enables multiple workgroups to be defined across switches and offers traffic segmentation and access control).

WAN Interface Connector

(WIC) A daughter card arrangement on the 3600 and 2600 series that allows a better mix of interfaces for slower speed WAN interfaces such as ISDN.

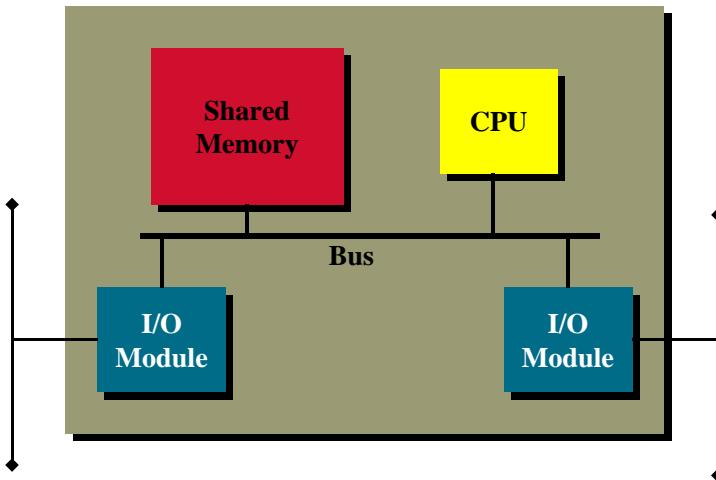
23.1 Switching, an Overview

Note This text is outdated, contact fc-uk@cisco.com for the updated CSSR (CEF Scalability and Selective Rewrite) information.

CSSR information. Regardless of platform, all routers have the same essential components. These include those in Figure 23-1:

- Interface modules—connect the router to the physical networks; transmit and receive packets
- Shared memory—stores packets as they enter and leave the router; buffers packets; also called “packet memory”
- CPU—performs packet processing, table maintenance, etc. tasks; has its own memory for storing configuration information, caches, tables, and the IOS image
- Single bus or multiple buses—connect these components together

Figure 23-1 Essential Components of All Routers



The first thing that all routers do upon receipt of a packet is to remove the layer-2 frame information and store the encapsulated layer-3 packet in packet memory. What happens to the packet from this point on is dependent upon the switching path that the packet is to follow. During the history of the IOS software, a number of switching paths have been developed and they are highly platform-dependent:

- Process Switching
- Fast Switching
- Autonomous Switching
- Silicon Switching
- Optimum Switching
- Distributed Switching
- NetFlow Switching
- Cisco Express Forwarding (CEF) & Distributed CEF (dCEF)

In addition to being platform-dependent, the paths available on a specific platform can be version-dependent. That is, the version of IOS software being used can be a factor. For example, most low-end and mid-range platforms, such as 25xx, 36xx, and 16xx, only supported Process Switching and Fast Switching in Releases 11.0, 11.1, 11.2, and 11.3. But Release 12.0 saw CEF support extended to the low- and mid-range platforms (although the 16xx range was still not supported at the time of this writing, April 2000).

Note CEF support for the various platforms has appeared throughout the development of the 12.x trains. For example, CEF support for the AS5300 appeared in 12.0(5.5)T. For specific platform requirements, check the relevant resource for IOS requirements.

One instance of integrating CEF switching into a specific application is described in the IOS Technical Note, “VP CEF Support,” available off <http://wwwin-enged.cisco.com/ios/doc>.

Because of the platform and version dependency of switching paths, the rest of this overview is organized as follows:

- Low-End and Mid-Range Systems
- 7000-Specific Switching Paths
- Switching Paths on High-End Platforms
- Load Balancing and Other Features

23.1.1 Low-End and Mid-Range Systems

For purposes of this chapter, low-end and mid-range systems are classified as those having a single CPU and no ability for distributed processing or intelligent line card support. Examples of platforms that fall into this category are:

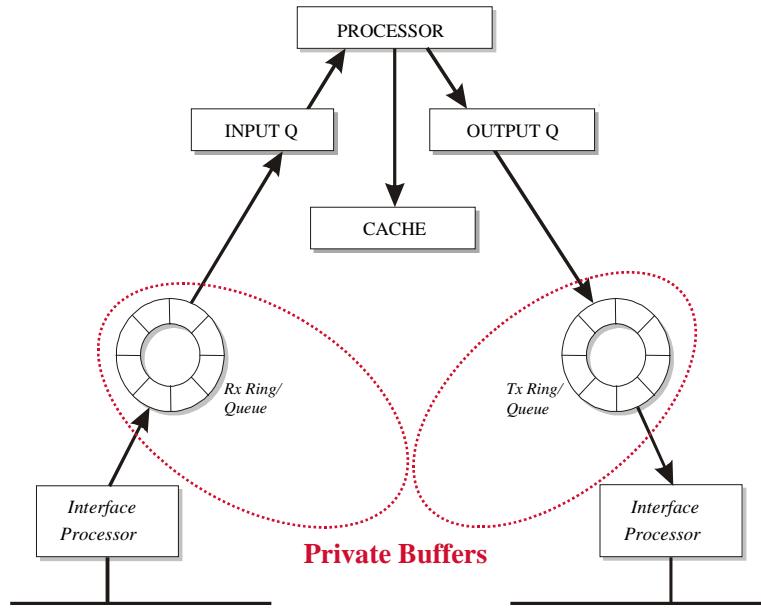
- 16xx
- 25xx
- 36xx
- 4x00
- 5x00

As noted previously, until the 12.0 IOS code, only two switching paths were available for these platforms: Process Switching and Fast Switching. With 12.0, CEF and dCEF became available.

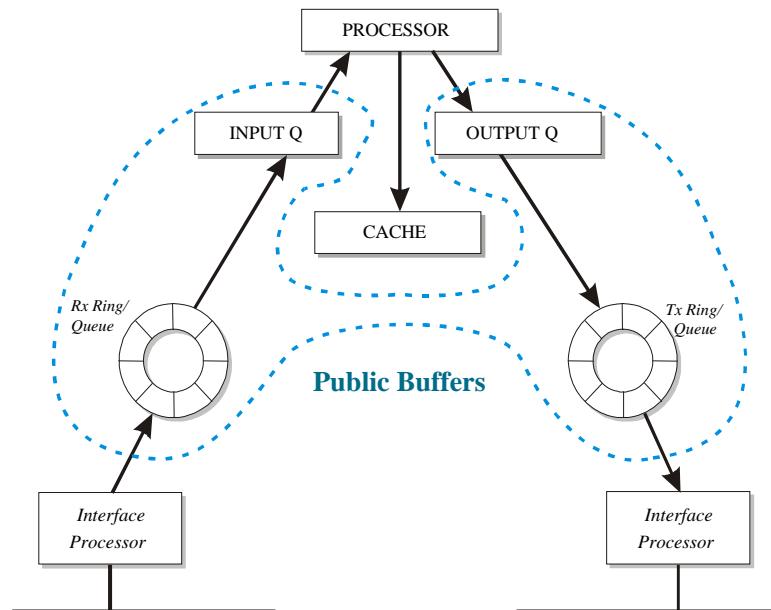
Within low-end and mid-range systems, the packet flow through any router, regardless of switching mechanism, is closely related to buffer usage. There are two basic buffer pools available: private buffers, Figure 23-2, and public buffers, Figure 23-3.

Some network interface processors create pools of private buffers when they initialize, some do not. Private buffer pools can be viewed with the **show buffers** command. Other useful **show** commands can be found at:

http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/ffun_c//fcfprt3/fcf012.htm#1023527

Figure 23-2 Private Buffers

Public buffer pools are created by the IOS system and are used to process switch packets. They are also used by network interfaces that either run out of private buffers or do not support the private buffer function.

Figure 23-3 Public Buffers

When a packet first arrives on an interface, it is placed in a buffer on the receive ring (**Rx Ring/Queue** in Figure 23-3). The interface processor then tries to replace this used buffer with a free buffer, either from its private pool or, if this is not possible, with a buffer from the public pool.

If the packet is to be process switched, then ownership of that buffer passes from the interface processor to the CPU. If the packet is to be fast switched, ownership passes to either the output queue or the outbound transmit ring (Tx Ring/Queue).

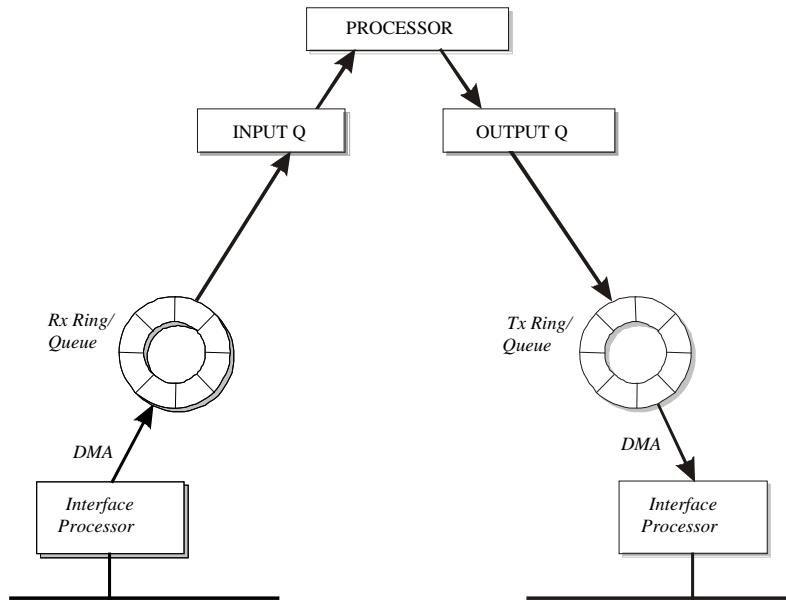
It is important to note that in low-end and mid-range systems, packets are never copied from buffer to buffer. Only the *ownership* of the buffer changes, via the use of pointers.

Once the packet has been transmitted, the buffer is returned to its original owner.

23.1.1.1 Process Switching

The term “process switched” refers to the fact that the CPU is directly involved in the decision process required to forward the packet at *process level*. This is as opposed to fast switching which, although still requiring the direct involvement of the CPU, occurs at *interrupt level*. Process switching has also been called “slow switching” and “routing.”

Figure 23-4 Process Switched Path



- 1 After a packet arrives on an inbound interface, the interface driver must first copy that packet (“DMA” in Figure 23-4) into a packet buffer in shared memory. This packet buffer could be pulled from either a public or private pool and is done without signaling an interrupt to the CPU.
- 2 The interface driver next determines what type of layer-3 protocol is encapsulated in the packet. This information is also buffered.
- 3 Once the interface driver has buffered the packet and identified the layer-3 protocol, it generates an interrupt to the CPU (“Processor”), indicating that a packet is waiting for processing in the input queue (“INPUT Q”).
- 4 Once the processor receives the interrupt generated by the interface driver, it assumes ownership of the packet buffer, determines which process must be called to handle this packet, and then schedules that process to run.

At this point, there is some period of “idle time” for the packet as it waits for the called process to be run. Exactly how much idle time is dependent upon the number of outstanding processes waiting to run, the number of additional packets waiting to be forwarded, etc.

- 5 When the process that handles the packet type finally runs, it determines which interface this packet should be forwarded out of by doing a route table lookup. If it determines that the packet is destined for the router itself, the packet will be requeued for additional processing. If it determines that this packet is to be forwarded, a new layer-2 header will be added to the packet and it will be placed on the relevant output queue (“OUTPUT Q”).
- 6 The process that handles the destination output queue will then place the packet onto the interface transmit ring (“Tx Ring/Queue”). The interface driver identifies that there is a packet in the transmit ring waiting to be sent and forwards it out onto the physical media. The interface driver then signals an interrupt back to the processor, requesting that counters be updated and buffers be placed back into free pools.

These tasks are divided among three processes in this way:

- 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 processes all packets destined for the router itself (except for routing updates: the routing updates for that layer-3 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.

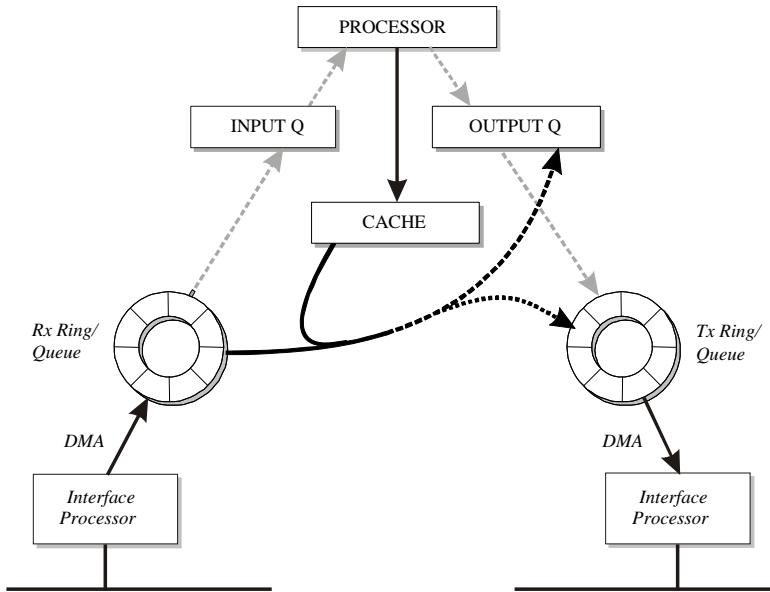
More than one process for a protocol can be active at the same time. For example, a single router might be running BGP, IP Enhanced IGRP, and IP RIP—all IP routing protocols—at the same time.

23.1.1.2 Fast Switching

Fast switching is a method of performing a routing lookup and forwarding a packet from the interrupt level. It thereby avoids queueing the packet for a process, the latency of scheduling the process, and any latency within the process itself. Fast switching relies upon a special lookup table that is maintained in processor memory by the individual routing processes. This table (“CACHE” in Figure 23-5) contains destination layer-3 addresses, corresponding layer-2 addresses, and the associated outbound interfaces.

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 critical because it runs at interrupt level.

Although Fast Switching is applicable to all IOS platforms, it does not support all protocols or packet features. For example, TCP header compression requires CPU processing. And some IBM and X.25/LAPB protocols cannot be fast switched.

Figure 23-5 Fast Switched Path

The first and second steps in the Fast Switching path are the same as in the process switching path.

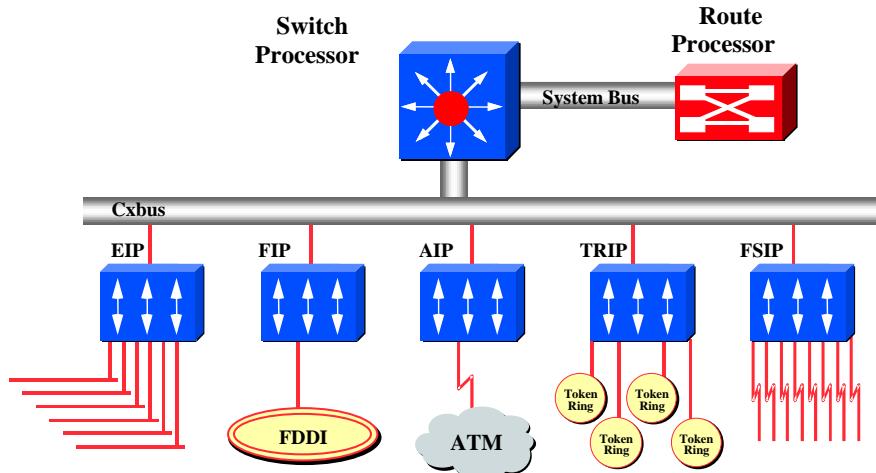
- 1 After a packet arrives on an inbound interface, the interface driver first copies the packet into a packet buffer in shared memory. This buffer could be pulled from either a public or a private pool and is done without interruption to the CPU.
- 2 The interface driver next determines what type of layer-3 protocol is encapsulated in the packet. This information is also buffered.
- 3 The interface driver then examines the switching cache to determine whether or not an entry exists for the required destination.
- 4 If a valid entry exists, the new layer-2 header will be copied from the cache and prepended onto the layer-3 packet. Then, using information from the cache, the interface driver determines which outbound interface should be used for forwarding this packet.
- 5 If the outbound interface already has packets in its outbound queue (“OUTPUT Q” in Figure 23-5), the driver will add the new packet to the end of the queue. If the queue is empty, the driver will place the new packet directly onto the transmit ring (“Tx Ring/Queue”).
- 6 After successful transmission onto the physical media, the transmitting interface processor signals a transmit interrupt to the processor, so that counters can be updated, buffers returned, etc.

If the cache had not contained a valid entry for the required destination, the interface driver would have signaled a receive interrupt to the processor and the packet would have been process switched (described in the previous section, 23.1.1.1). However, in addition to forwarding the packet, the processor would have used the results of the forwarding decision to populate the fast cache, so that subsequent packets to the same destination could be fast switched.

23.1.2 7000-Specific Switching Paths

The Cisco 7000 router has two additional switching mechanisms, known as Autonomous and Silicon Switching. As Figure 23-6 shows, the 7000 router is constructed from a Route Processor and a Switch Processor.

Figure 23-6 7000 Router Switch Path



The Route Processor maintains the routing table as well as the fast-cache table. As one can see from the diagram above, communication between the Route Processor and the Cxbus requires the involvement of the Switch Processor. The additional switching methods enable caches within the Switch Processor, reducing or removing the need to communicate with the Route Processor.

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, a subset of which is maintained by the Route Processor. With Autonomous Switching enabled, once the fast-cache has been populated and the information propagated to the Autonomous cache, packets can be "routed" without the need for communication with the Route Processor.

Silicon Switching is an enhanced form of Autonomous Switching. An additional hardware module known as the SSE (Silicon Switching Engine) is added to the Switch Processor. The SSE has a larger cache than the Autonomous cache and is therefore capable of holding more entries. It also organizes the fast-cache structure into a more efficient form, leading to faster lookup times. As with Autonomous Switching, synchronization is maintained between the Route Processor and the SSE.

23.1.3 Switching Paths on High-End Platforms

This subsection deals with the high-end router platforms. The devices that are considered high-end include the 75xx series, 720x/71xx series, and the Gigabit Switch Router (GSR) 120xx series.

Table 23-1 lists the switching paths available for each of these platforms. Following the table, each of the switching paths listed is described.

Table 23-1 Switching Options on High-End Routers

Switching Path	Cisco 7000 w/RSP	Cisco 720x/71xx	Cisco 7500	Cisco GSR 120xx	Comments
Process Switching	Y	Y	Y	Y	Initializes switching caches
Fast Switching	Y	Y	Y	N	Default (except for IP, which is Centralized CEF for v12.0 onward)

Table 23-1 Switching Options on High-End Routers (continued)

Optimum Switching	Y	Y	Y	N	Default for IP (pre-v12.0)
NetFlow Switching	Y	Y	Y	Y	Configurable per interface (GSR depends on IOS version)
Distributed Optimum Switching	Y	N	Y	N	Using VIP2-20/VIP2-40/VIP2-50 (<i>not available in v12.0 but can be configured!</i>)
Centralized CEF	Y	Y	Y	N	Default for IP (v12.0 onward)—except 72xx/71xx
Distributed CEF	N	N	Y	Y	Only on 75xx with VIPs & GSR platform

23.1.3.1 Route Switch Processor Architecture Overview

The Route Switch Processor (RSP) is used in the 7000/RSP and 75xx series. It is important to have a clear understanding of the RSP architecture and the components used to switch a packet through an RSP.

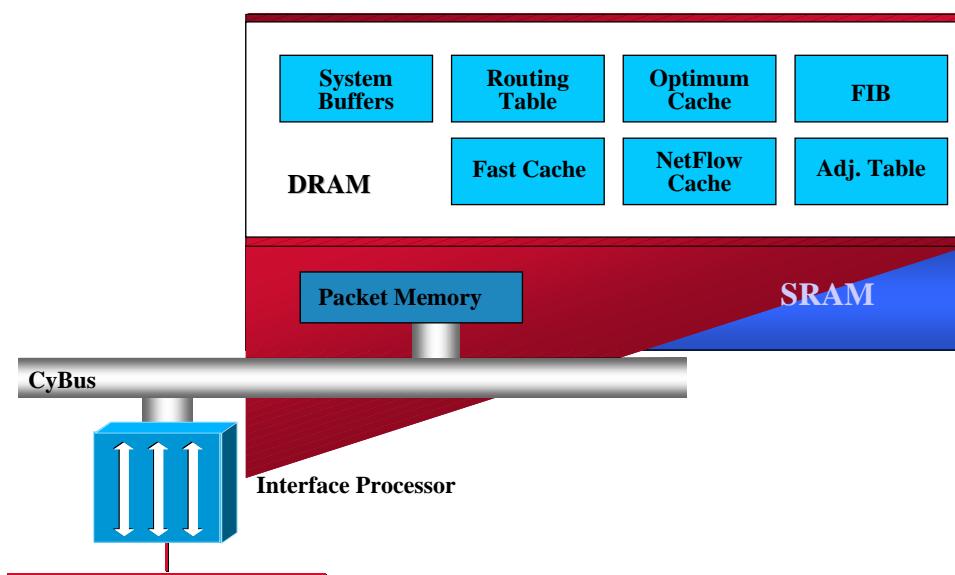
Figure 23-7 RSP Memory Allocations

Figure 23-7 shows the following components.

- Cybus

All RSP platforms are based on a bus architecture. The RSP, Interface Processors (IPs), and Versatile Interface Processors (VIPs) are interconnected via a 32-bit-wide bus.

- SRAM

SRAM, also known as MEMD, is a 2-megabyte memory block found on *all* versions of RSP (that is, RSP1, 2, and 4). With the exception of VIP local switching, all packets received by the router will be buffered in RSP SRAM.

The 2 megabytes of SRAM is carved between the various interface cards present in the system into different sized buffer pools using a complex carving algorithm. The results of this carving algorithm can be seen with the output of a **show controller cbus** command. It should be noted that the carving process will reserve buffers for an interface even if none of the interfaces on the line card are configured for use.

- DRAM

DRAM is physically located on the RSP itself. Depending upon the model, the maximum onboard can be either 128 megabytes (RSP1 & 2) or 256 megabytes (RSP4).

DRAM is used for the IOS software, storing cache tables, route tables, IPX SAP tables, etc. A **show process memory** gives a snapshot of DRAM usage by various processes.

System buffers also live in DRAM space. These buffers are used by all packets that arrive into the router and that are destined to be Processed Switched. Packets generated by the router also use system buffers. The command **show buffers** gives a current snapshot of system buffer utilization.

Regardless of the switching path being used, the procedure used by the router to deal with a packet as it is received on an interface is the same.

23.1.3.2 Packet Forwarding on Non-VIP-Based Line Cards

The following steps assume that the packet is being received on a legacy Interface Processor (IP), which includes cards such as the EIP-x, FIP-x, and HIP-x. (Packet switching using VIPs is discussed in 23.1.3.3, “Packet Forwarding on VIPs with Distributed Switching.”)

- The hardware processor detects that a packet has been received from the physical media and places it into one of the onboard receive buffers.
- The interface processor then tries to allocate a buffer from those available in SRAM.
 - First it tries to allocate a buffer from its own pool of allocated buffers in SRAM (local free queue).
 - If there are no available buffers in the local free queue, the interface processor will attempt to allocate a buffer from the global free queue.
 - If there are no buffers available in the global queue, the packet is flushed.
 - If the interface has already grabbed the maximum number of global buffers it is allowed, the packet is flushed.

Flushing the packet will lead to the ignore counter being incremented on a classical IP and RxSide buffering taking place on a VIP (that has sufficient free memory to support it).

- If there is a global queue buffer available, the packet is allocated that buffer.
- The interface processor then adds the buffer header from the newly buffered packet to a queue serviced by the CPU at interrupt level. As soon as the queue is non-empty an interrupt is signalled to the RSP CPU.
- At this point several things can happen, depending upon the switching path configured on the receiving interface.
 - If the input packet is an IP packet, and the receiving interface is configured for CEF, forward the packet using CEF.
 - If the input packet is an IP packet, and the receiving interface is configured for Optimum Switching, go to the Optimum Switching path. If the receiving interface is not configured for Optimum Switching, go to the Fast Switching path.

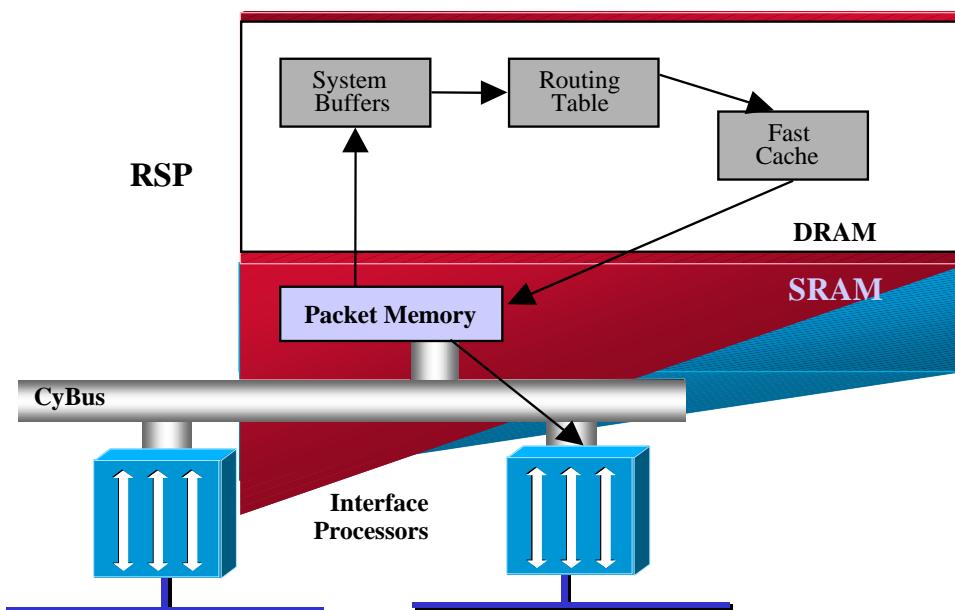
- If the input packet is IP and the receiving interface is configured for NetFlow Switching, go to the NetFlow Switching path.
- If the input packet is *not* an IP packet, go to the Fast Switching path.

23.1.3.2.1 Process Switching

When Process Switching, the following steps take place, assuming that the packet has already been allocated an SRAM buffer. Refer to Figure 23-8.

- The incoming packet is moved from an SRAM buffer into a DRAM buffer. The freed SRAM buffer is then returned to the local free queue of the receiving interface.
- If a DRAM buffer cannot be made available, or if the input hold-queue for the interface is full, the packet is dropped and the relevant counters incremented.
- However, if the packet is successfully allocated a DRAM buffer, it is then enqueued onto the relevant protocol input queue for further processing.
- If the packet is destined for the router itself (that is, broadcast, routing update, etc.), the packet is requeued for further processing.

Figure 23-8 RSP Process Switching



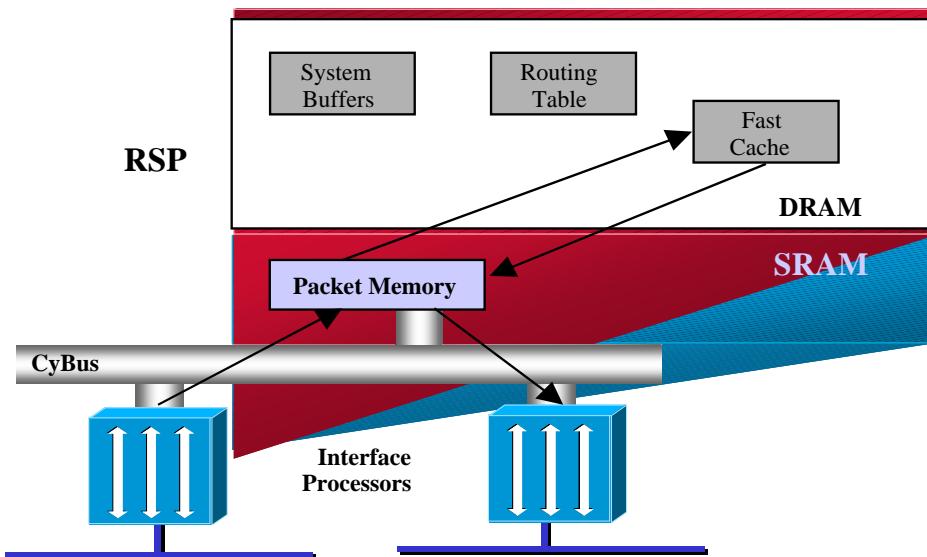
- If the packet is a unicast packet, the packet is forwarded depending on the longest available match in the routing table. The packet is then added to the relevant outbound interface output hold-queue. The RSP also adds a cache entry in the Fast Switching cache, so that subsequent packets to this destination can be fast switched.
- If the output hold-queue is full, the packet is dropped and the *output drop* counter is incremented.
- The output queue routine then dequeues that packet from the output hold-queue and places the packet in an SRAM buffer. From here, the packet is queued onto the outbound interface's transmit queue.
- The outbound interface processor retrieves the packet from the SRAM buffer and forwards out the physical interface. The SRAM buffer is then returned to either the local or global free queue.

23.1.3.2.2 Fast and Optimum Switching

Optimum Switching works in a *very* similar way to Fast Switching. The main difference between the two is that Optimum Switching uses a different cache tree which, while resulting in greater memory usage, also results in much faster cache lookup. When Fast Switching, the following steps take place, assuming that the packet has already been allocated an SRAM buffer. Refer to Figure 23-9.

- The RSP CPU checks the destination address of the packet in the SRAM buffer against fast cache entries.
- If the cache lookup is unsuccessful, the packet is then requeued for Process Switching.
- Assuming that the cache lookup results in a hit, the packet is rewritten with new layer-2 headers and placed back into the same RAM buffer.

Figure 23-9 RSP Fast Switching



- From here, the SRAM buffer is enqueued onto the relevant outbound interface transmit queue.
- If the interface outbound transmit queue is full, but the interface is not configured for *backing store*, the packet will be flushed and the output drop counter incremented. If the interface outbound transmit queue is full, but the interface is configured to use backing store, the packet is copied from the SRAM buffer into a DRAM system buffer. This results in the *output buffers swapped out* counter being incremented.

Note Backing store is enabled by default when using Weighted Fair Queuing and cannot be disabled.

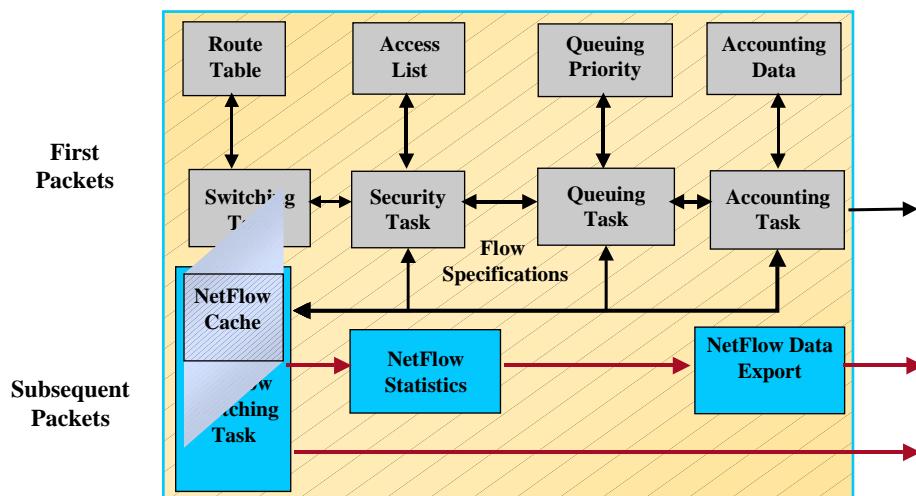
- The transmitting interface then dequeues the packet from the SRAM buffer into one of its own transmit buffers before forwarding it on the physical media.
- The SRAM buffer is then returned to the owning global or local free pool.

23.1.3.2.3 NetFlow Switching

NetFlow Switching provides network administrators with access to detailed accounting information from their data networks. It also provides a highly efficient mechanism with which to process extended or complex access lists without paying as much of a performance penalty as with other switching methods.

NetFlow Switching allows for Access Lists, Queuing, or Accounting policies to be applied to only the first packet in a flow. Assuming that the packet passes all required checks, a cache entry is created in the NetFlow cache known as a *flow tag* (not shown in Figure 23-10). The tag is created on the basis of the source and destination IP address together with the destination TCP or UDP port number. It should be noted that although most TCP/UDP can be considered bidirectional, flow tags are considered unidirectional. This means that if a conversation passes through the router between host1 and host2, two flow tags will be created.

Figure 23-10 NetFlow Switching



A *flow* is defined as conversation between the source and destination. For TCP communication, a conversation starts and stops with the various TCP control messages. For UDP conversations, a conversation is considered to have ceased after a timer has expired.

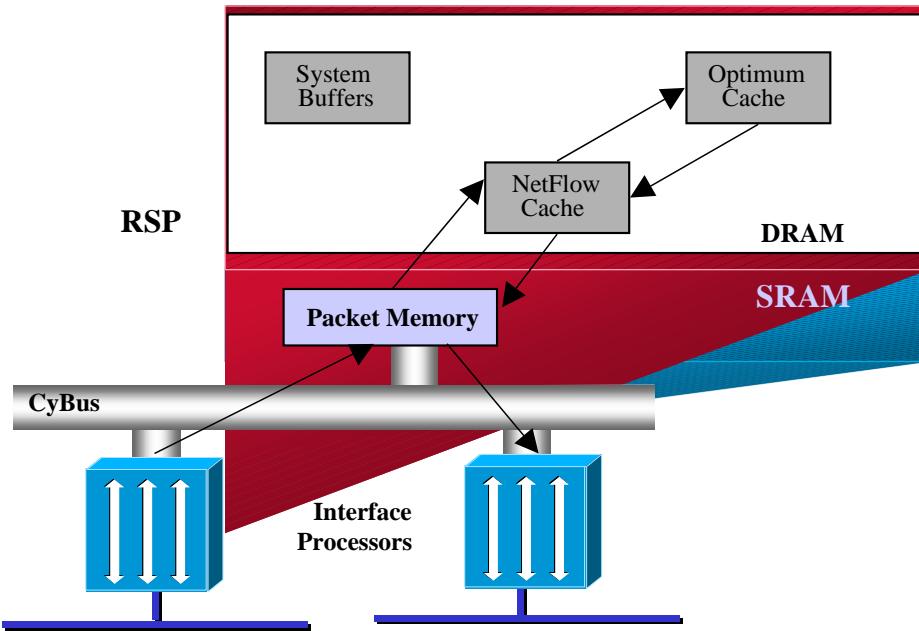
Subsequent packets that match the flow tag are deemed to be members of the same flow and are simply switched through the outbound interface, bypassing further checking against access lists, queuing, etc.

When NetFlow Switching, the following steps take place, assuming that the packet has already been allocated an SRAM buffer: Refer to Figure 23-11.

- The RSP CPU first checks the packet in the SRAM buffer against existing entries in the NetFlow cache table.
- If no match is found, the packet is passed on to traditional switching mechanisms (that is, Fast or Optimum Switching) for additional processing. As the packet passes along the Fast or Optimum Switching paths, any access list, queuing, or accounting policies implemented are used to populate the NetFlow cache with the packet flow information. If, after checking the NetFlow Cache, it is determined that the incoming packet is a member of a previously allowed traffic flow, the packet is rewritten with new layer-2 headers and placed back into the same SRAM buffer.
- *At the same time, any relevant or required accounting information is recorded, counters are incremented, etc.* From here, the SRAM buffer is enqueued onto the relevant outbound interface transmit queue.

- The transmitting interface then dequeues the packet from the SRAM buffer into one of its own transmit buffers before forwarding on the physical media.
- The SRAM buffer is then returned to the owning global or local free pool.

Figure 23-11 RSP NetFlow Switching



Note NetFlow is designed to offer performance enhancement in the parsing of extended or complex access lists. However, with simple or short access lists, due to the overhead imposed by NetFlow, it is possible that NetFlow Switching could lead to performance degradation.

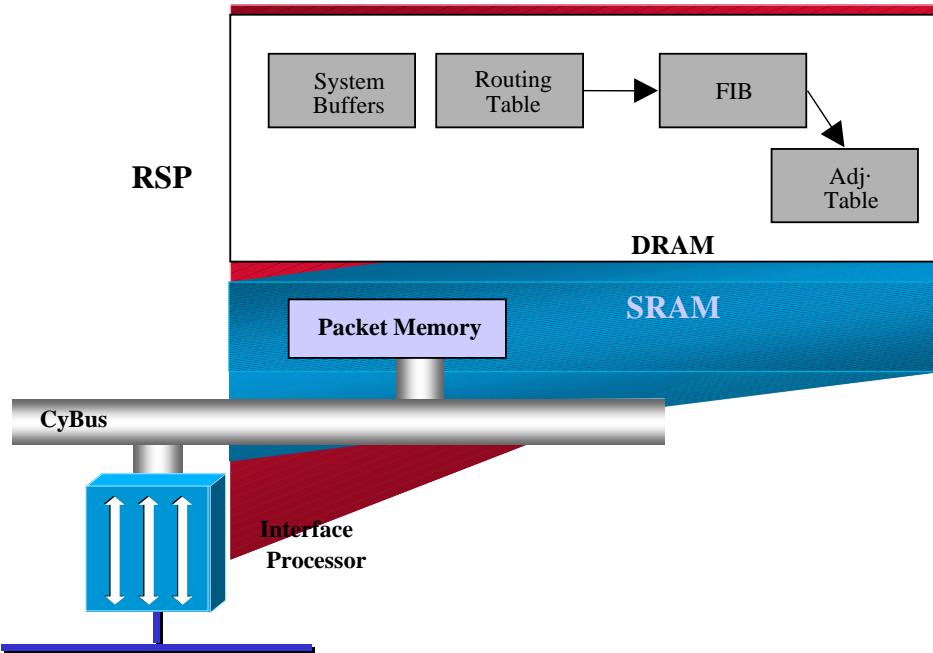
23.1.3.2.4 Cisco Express Forwarding (CEF)

The Demand-Based cache mechanisms discussed in the previous sections all suffer the following drawbacks:

- They are all traffic driven, in that they are dependent on receipt of the first packet to populate the cache.
- It is possible for caches to grow larger than routing tables, thus consuming significant amounts of memory.
- Periodic aging of the cache entries can consume large amounts of CPU time if the cache is large.
- Cache invalidation due to a route-flap relies on Process Switching to repopulate the cache with valid entries.
- On some platforms, the size of the cache may lead to cache entry churn if there are too many entries for the cache to support.
- Per-packet load balancing cannot be done from interrupt level.

It was the inherent drawbacks to traditional demand-based caches that led to the development of Cisco Express Forwarding (CEF) and Distributed CEF (dCEF, discussed later in 23.1.3.3, “Packet Forwarding on VIPs with Distributed Switching,” 23.1.3.7, “Distributed Switching Compatibility Matrix,” and other sections).

Figure 23-12 RSP, CEF Forwarding Tables



The two main components of CEF are the Forwarding Information Base (FIB) and the adjacency table. Both tables are stored in DRAM memory, as shown in Figure 23-12.

The FIB table is used to make IP destination prefix-based forwarding decisions. It contains a mirror image of the information stored in the IP routing table. When routing or topology changes occur in the network, the IP routing table is updated and those changes are reflected in the FIB. The FIB maintains next-hop address information based on the information in the IP routing table.

CEF also uses adjacency tables to prepend layer-2 addressing information. The adjacency table maintains layer-2 next-hop addresses for all destination layer-3 FIB entries. The entries allow the RSP to perform fast layer-2 header rewrites when switching the packet from source interface to destination interface.

The adjacency table is populated as adjacencies are discovered. Each time an adjacency entry is created (such as through the ARP protocol), a link-layer header for that adjacent node is precomputed and stored in the adjacency table. Once a route is determined, it points to a next hop and corresponding adjacency entry. That entry is then subsequently used for encapsulation during CEF switching of packets.

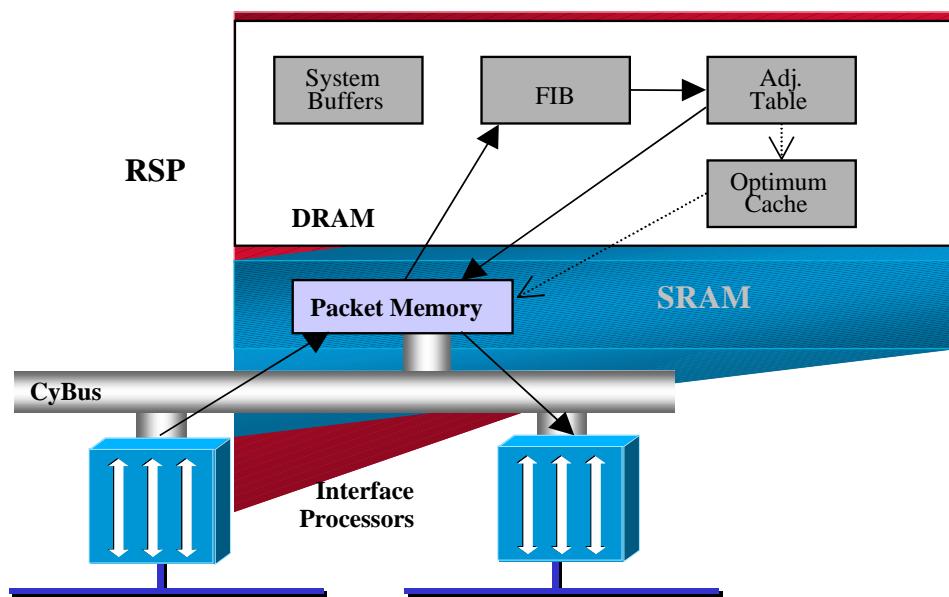
A full functional description of CEF and its components is outside the scope of this chapter. However, further information can be found at the following URL:

<http://www.cisco.com/univercd/cc/td/doc/product/software/ios112/ios112p/gsr/cef.htm#xtocid262647>

When CEF Switching, the following steps take place, assuming that the packet has already been allocated an SRAM buffer. Refer to Figure 23-13.

- The RSP CPU checks the destination address of the packet in the SRAM buffer against CEF destination entries stored in the FIB database.
- If the FIB lookup is unsuccessful, that is, the destination route is not present in the table or CEF does not support the received packet type, then one of two things takes place. The packet is either dropped (and no further switching of the packet takes place) or it is passed on to the next switching level, if one is available.
- If the cache lookup results in a hit, the corresponding adjacency entry is checked. If the adjacency entry is “good,” the packet is rewritten with the layer-2 headers that have been precomputed in the adjacency table. It is then placed back into the same SRAM buffer.

Figure 23-13 RSP CEF Switching



- If the destination prefix points to a “punt” adjacency, the packet is requeued for further processing by the next switching path, typically Optimum Switching.
- From here, the SRAM buffer is enqueued onto the relevant outbound interface transmit queue.
- If the interface outbound transmit queue is full, but the interface is not configured for backing store, the packet will be flushed and the “output drops” counter incremented.
- If the interface outbound transmit queue is full, but the interface is configured to use backing store, the packet is copied from the SRAM buffer into a DRAM system buffer. This results in the output buffers swapped out counter being incremented.

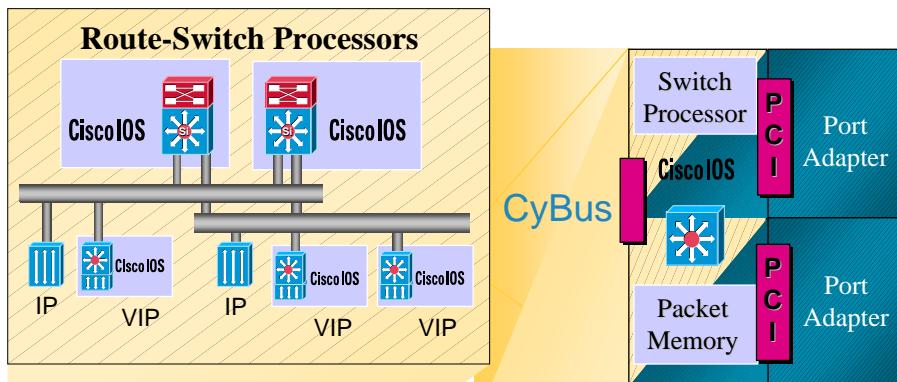
Note Backing store is enabled by default when using Weighted Fair Queuing and cannot be disabled.

- The transmitting interface then dequeues the packet from the SRAM buffer into one of its own transmit buffers, before forwarding it on the physical media.
- The SRAM buffer is then returned to the owning global or local free pool.

23.1.3.3 Packet Forwarding on VIPs with Distributed Switching

A Versatile Interface Processor (VIP) card has its own CPU, DRAM, and SRAM. Refer to Figure 23-14. (For additional VIP architecture information, see separate VIP documentation in EDCS.)

Figure 23-14 VIP Overview



The VIP SRAM is like the RSP SRAM and is used for storing packets. However, unlike RSP SRAM, VIP SRAM is not carved up into various MTU-sized buffers. All VIP SRAM buffers are of an equal size, 512 bytes. These buffers are also referred to as *particles*.

Distributed Switching is a mechanism that offloads packet forwarding decision making from the RSP CPU and moves it onto the CPU on the VIP card. Each VIP card maintains a copy of the CEF/switching cache and attempts to make forwarding decisions locally without signalling an interrupt to the RSP CPU.

Currently, Distributed Switching is only available for IP traffic and comes in three flavours;

- Distributed Optimum/Fast Switching
- Distributed NetFlow
- Distributed CEF (dCEF)

Note In release 12.0 of the IOS software, although Distributed Switching can be configured, in some cases it only has the effect of enabling centralized Optimum Switching. The only Distributed Switching mechanism supported in 12.0 is dCEF. Distributed Netflow works in the same manner as centralized Netflow, except that the processing of packets and the maintenance of the flow tag caches are distributed to the VIP.

23.1.3.3.1 VIP Packet Forwarding Using dCEF and Distributed Switching

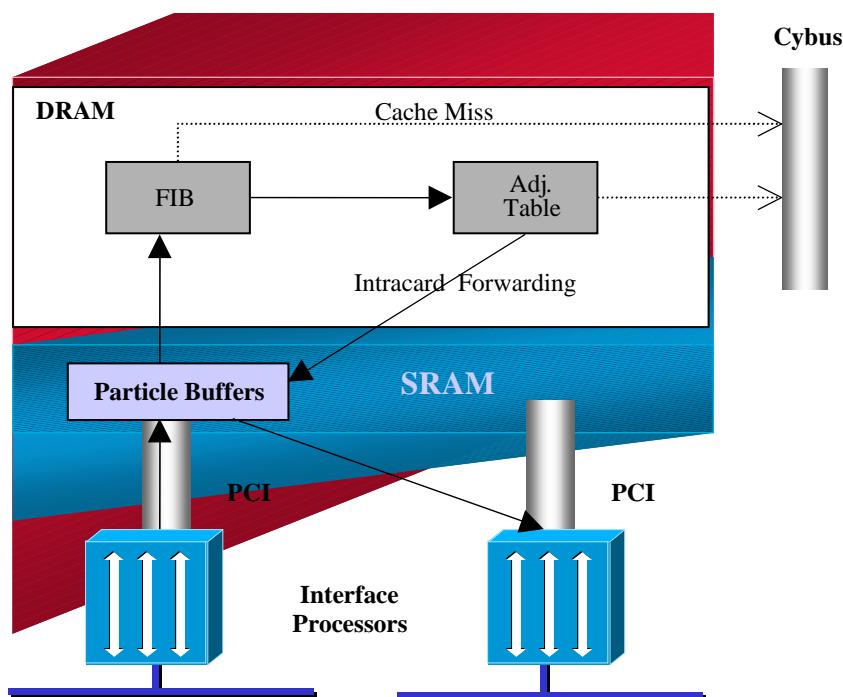
Although these two switching paths are separate entities and use different tables and so on, they operate in a similar fashion with regard to forwarding a packet through the router. It should be remembered that if dCEF is enabled, copies of the FIB and adjacency tables are created on the VIP even if CEF/dCEF is disabled at the interface level. It is not possible to turn dCEF off on a particular VIP.

When dCEF switching on Cisco VIPs, the following steps take place. Refer to Figure 23-15.

- The port adaptor detects an incoming frame and attempts to allocate a sufficient number of particle buffers in VIP SRAM. If a sufficient number of particles are not free, the frame is flushed and the interface *ignore count* is incremented.

- Assuming that there are sufficient particle buffers available, the packet is buffered and an interrupt is sent to the VIP CPU.
- The VIP CPU at interrupt level tries to switch the packet by looking up its destination in the relevant distributed cache or dCEF table held in VIP DRAM. If the VIP lookup does not result in a cache hit, the VIP then tries to allocate a buffer in RSP SRAM and pass the packet onto the RSP for further processing. The packet is then switched in the same manner as if it had been received on a legacy IP (see “Packet Forwarding on Non-VIP-Based Line Cards”).
- If the VIP lookup results in a cache hit, and the outbound port is on the same VIP, the packet is locally switched (*intracard forwarding*). The packet is queued on the outbound queue of the relevant port adaptor. With local switching, performance is optimized because the packet does not need to cross the Cybus.

Figure 23-15 dCEF Switching on Cisco VIPs



- If the VIP lookup results in a hit, but the destination interface is on another IP or VIP (*intercard forwarding*), the VIP requests an RSP SRAM buffer from its local free queue. However, if the VIP's local free queue is empty, the VIP then requests a buffer from the global free queue. There are four possible outcomes to this request.
 - If there are no global buffers available, and the VIP does not support *receive buffering*, the packet is dropped.
 - If there are no global buffers available, but the VIP does support receive buffering, the VIP buffers the packet internally and will rerequest a RSP SRAM buffer.
 - If there are global buffers available but the outbound TxQueue is full, and the VIP does support receive buffering, the VIP buffers the packet internally and will rerequest an RSP SRAM buffer.
 - If there are sufficient buffers available, the packet is buffered successfully on the transmit queue of the outbound interface.

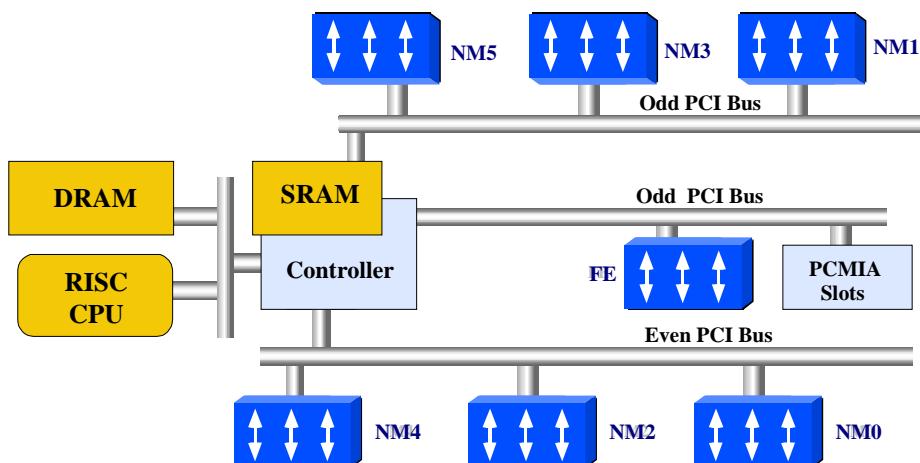
- The transmitting IP/VIP then dequeues the packet from the RSP SRAM buffer for forwarding out on the physical media.
- Freed buffers are then returned to the pools of the buffer's original owner. That is, particle buffers will be returned to the owning VIP's particle buffer pools and MTU-based buffers will be returned to the local free queue buffer pools.

23.1.3.4 Packet Forwarding on 720x Platforms

23.1.3.4.1 Packet Forwarding on 7206VXR/NPE Route Processor Cards

The 7200 has a variety of Route Processor cards available. They are all based on the Mips Rx000 series processor. The 7200 can be thought of as the chassis form of a VIP card in that much of the architecture of the 7200 is common to the VIP. See Figure 23-16.

Figure 23-16 7200 Architecture



- **150MHz - 260MHz RISC Processor**
- **2 PCI Buses**
- **SRAM for Fast Interfaces Support**

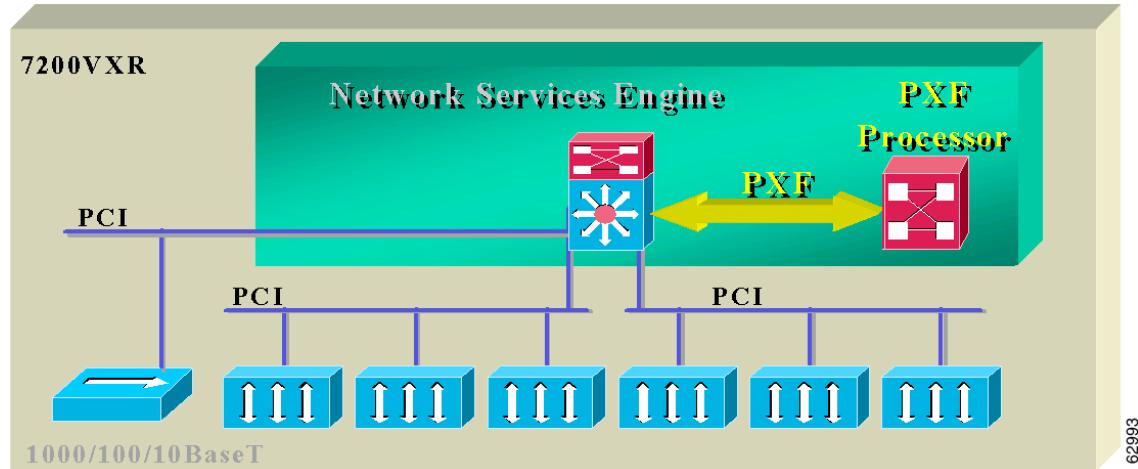
The 7200 is capable of all of the switching paths found on RSP-based platforms, with the exception of Distributed Switching. The 7200 uses the same port adaptors as the VIPs but uses a single centralized CPU. Therefore, it is not possible to offload any processing.

Note All previous discussions on VIP-based centralized switching can be applied to the 7200 platform.

23.1.3.4.2 Packet Forwarding on 7200VXR / NSE-1 Processor Cards

The Network Services Engine-1 (NSE-1) is a processing engine that can be inserted in the 7200VXR chassis, and the architecture is similar to that found in the normal 7200. The major exception is the introduction of the Parallel eXpress Forwarding (PXF) Processor. See Figure 23-17.

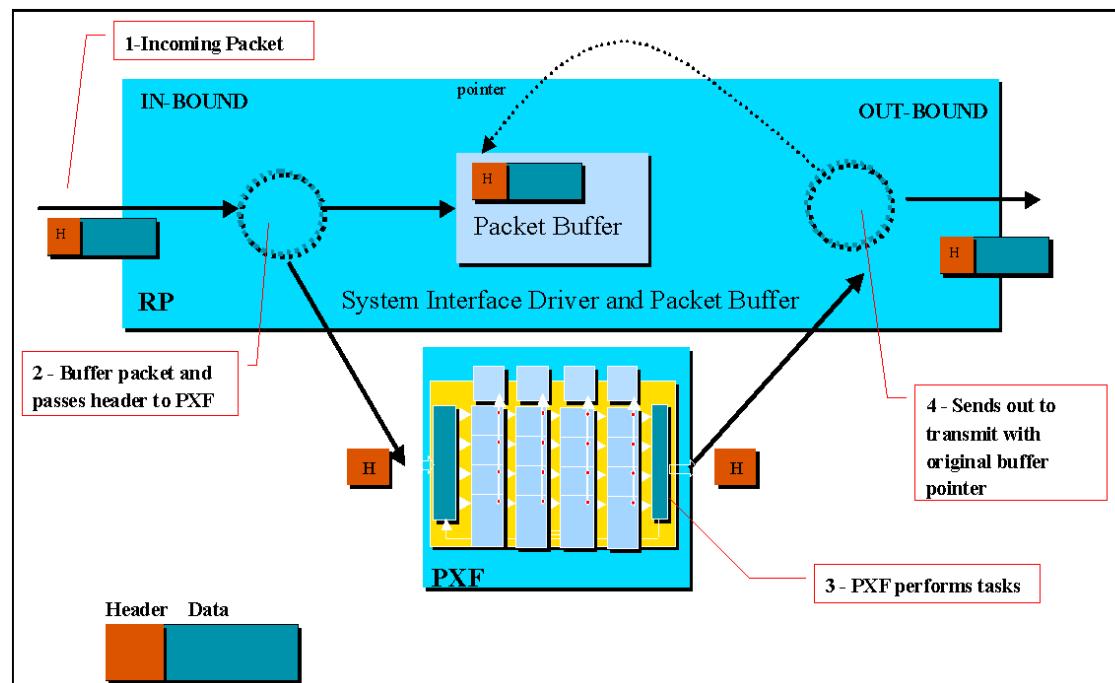
Figure 23-17 NSE 1 Architecture



**Embedded PXF Processor for service acceleration
RM7000 Processor as per NPF300**

The issue that the PXF hardware is designed to address is that, as packet-affecting features such as Network Address Translation (NAT), Access-Control Lists (ACLs), or NetFlow switching are applied to the traffic stream, raw packet processing performance typically degrades. The PXF hardware is designed to take the processing of such features away from the CPU and have them handled via dedicated hardware.

Figure 23-18 NSE 1 Packet Path (1)



The major difference with the introduction of the PXF is that the majority of processing occurs in the PXF hardware, without requiring intervention from the Route Processor (CPU).

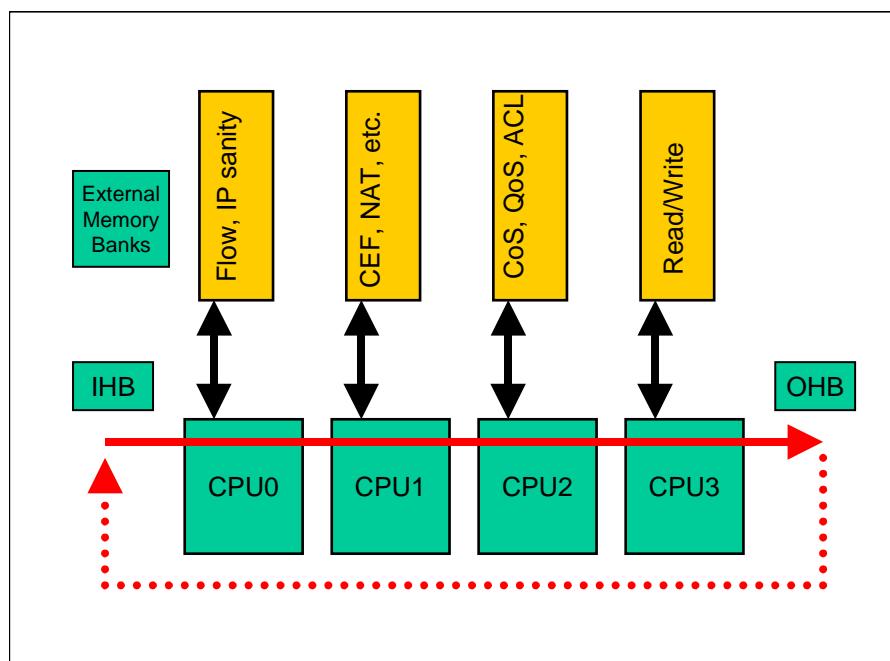
The incoming packet is received by the port adaptor and placed into a system buffer, as shown in Figure 23-18. The packet header (IP,TCP,UDP,ICMP) is read from this buffer and fed into the PXF via “packet feeder” hardware. The PXF also receives the address pointer of the buffer where the packet is stored. The PXF then performs tasks such as CEF lookup, L2 rewrite, and feature processing (if supported). Once completed, the PXF passes the processed header and buffer pointer to the RP. The RP then has all the necessary information required to send the packet out of the correct interface.

If the packet needs to be processed centrally, the PXF will “punt” the packet to the RP. Equally, if the particular feature that is meant to be applied to the packet is not supported by the PXF, the PXF will again punt the packet up to the RP.

CEF information, NAT data, Access-Control Lists, etc. are all calculated and compiled on the RP. The processed data is then copied to the PXF hardware. Changes such as CEF adjacencies are synchronized in a similar manner to that found on the GSR platform.

The PXF itself contains 16 100-MHz processors arranged into a 4x4 matrix. A single row of processors is responsible for handling a single packet. Processors in a single column are tasked with performing a particular task.

Figure 23-19 NSE 1 Packet Path (2)



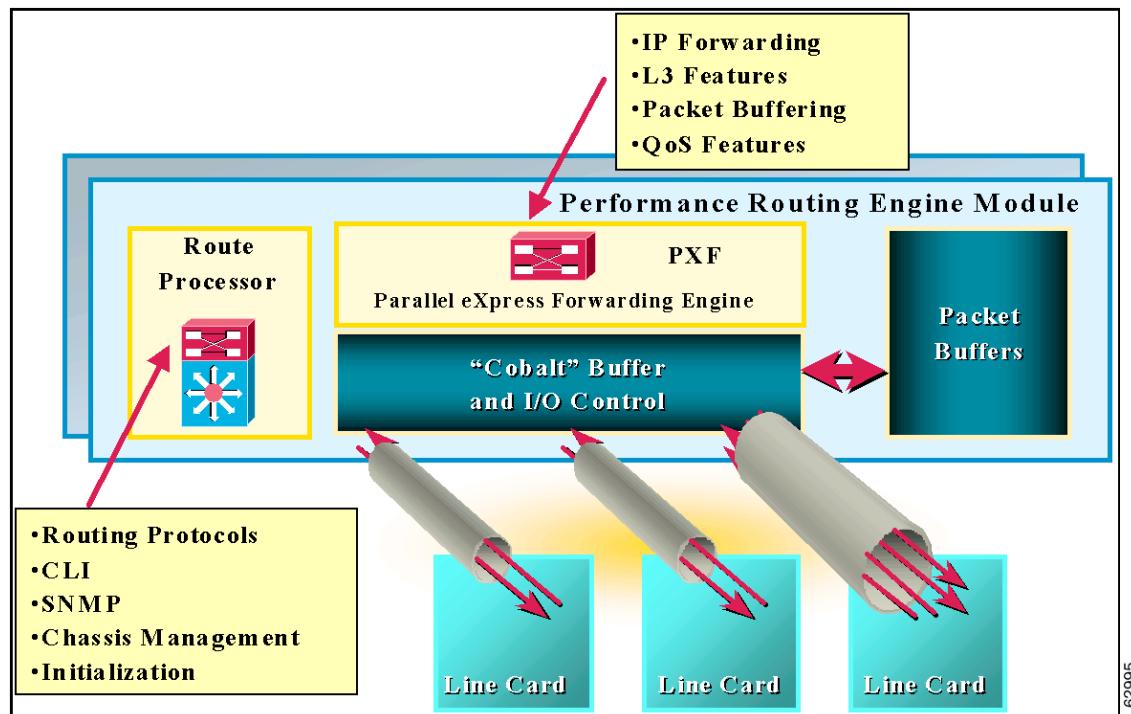
In Figure 23-19, you can see that the CPU0 column handles Netflow and IP CRC control, whereas the CPU1 column applies CEF and NAT functions.

At startup, a single packet is read into the first row. Once a header has been processed in column 0, it is then passed to column 1. In the meantime, a new packet header is placed into column 0 of row 2. Packets are read and processed in parallel; they “clocked” into the PXF array sequentially. Therefore, to fill the CPU pipelines in all four rows, four packets must be clocked into the PXF.

23.1.3.5 Packet Forwarding on the ESR 10000

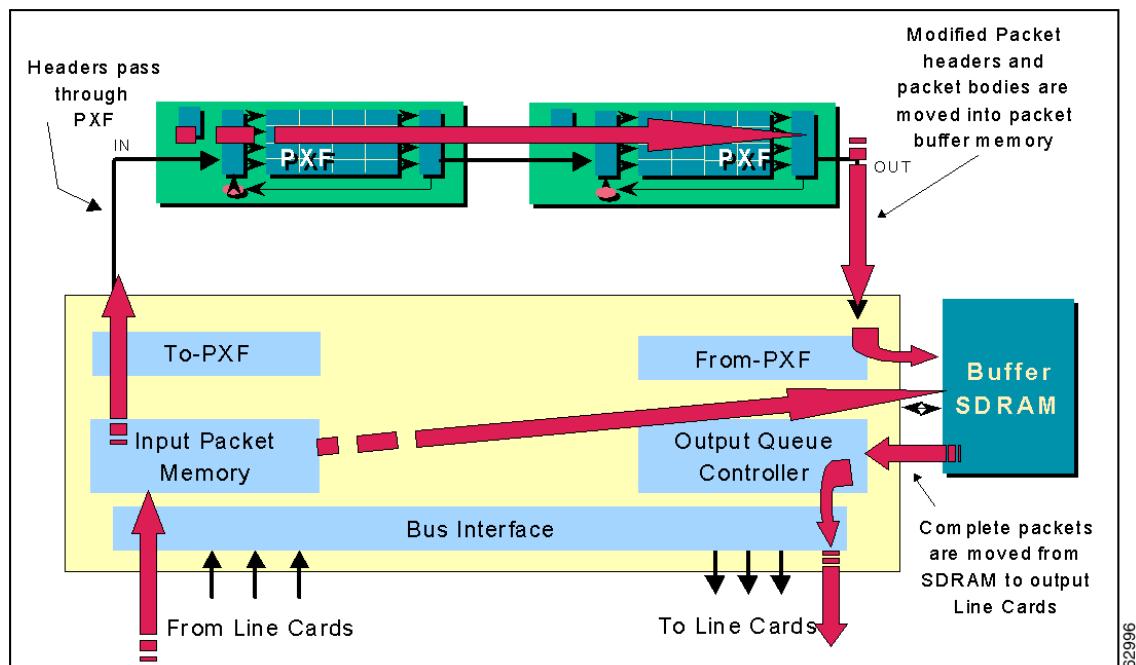
The ESR 10000, as shown in Figure 23-20, builds on the Parallel eXpress Forwarding (PXF) processors seen in the NSE-1. As in the NSE-1, the PXF hardware is responsible for packet header processing, while the Route Processor (CPU) is responsible for the routing protocols, chassis management, and for ensuring that the processed packets are sent out to the correct interfaces.

Figure 23-20 ESR 10000 Architecture



Packets arriving from the line cards are copied into input packet buffers. The header is copied into the packet feeder hardware for processing by the PXF, while the body of the packet is copied to buffers.

Figure 23-21 ESR 10000 Packet Flow (1)

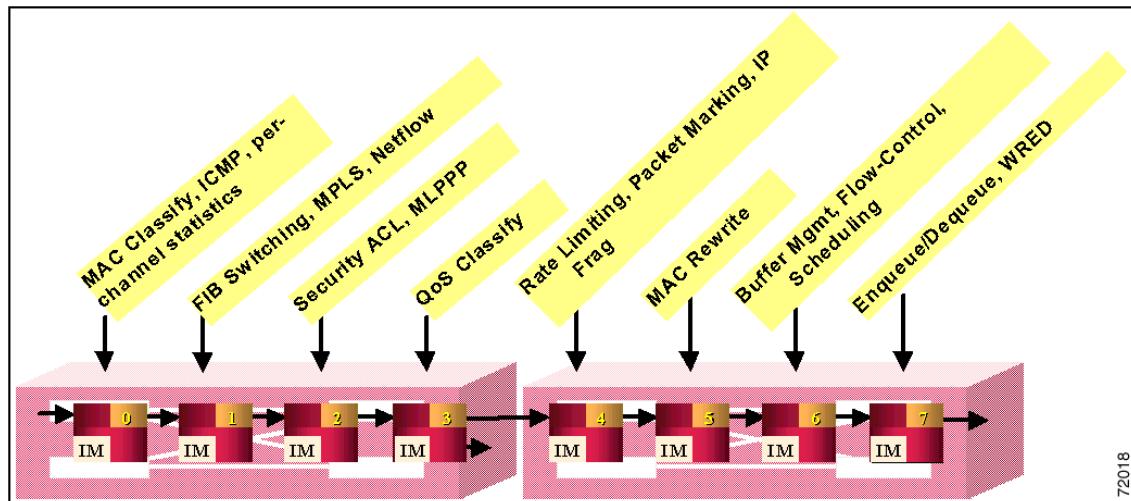


Packet headers pass into the first PXF. The ESR 10000 contains two PXF processing engines, as shown in Figure 23-21. This produces two 4 x 4 processing matrices. Each of the eight resulting columns is responsible for a particular set of tasks. Again, as per the NSE-1, access lists, CEF information, and other packet-affecting features are processed centrally on the RP and transferred to the PXFs.

Once the processing has been completed, the modified header is matched back to the packet body before being queued for transmission. If a packet cannot be PXF processed, the header is passed to the RP.

As mentioned above, the PXF complex in the ESR 10000 differs from the NSE-1 in that there are two sets of processors forming a complex of 32 processors organized into 4 rows of 8 processors. Each column serves a particular function.

Figure 23-22 ESR 10000 Packet Flow (2)



72018

At startup, a single packet is read into the first row, as shown in Figure 23-23. Once a header has been processed in column 0, it is then passed to column 1. In the meantime, a new packet header is placed into column 0 of row 2. Packets are read and processed in parallel; they are “clocked” into the PXF array sequentially. Therefore, to fill the CPU pipelines in all four rows, four packets must be clocked into the PXF.

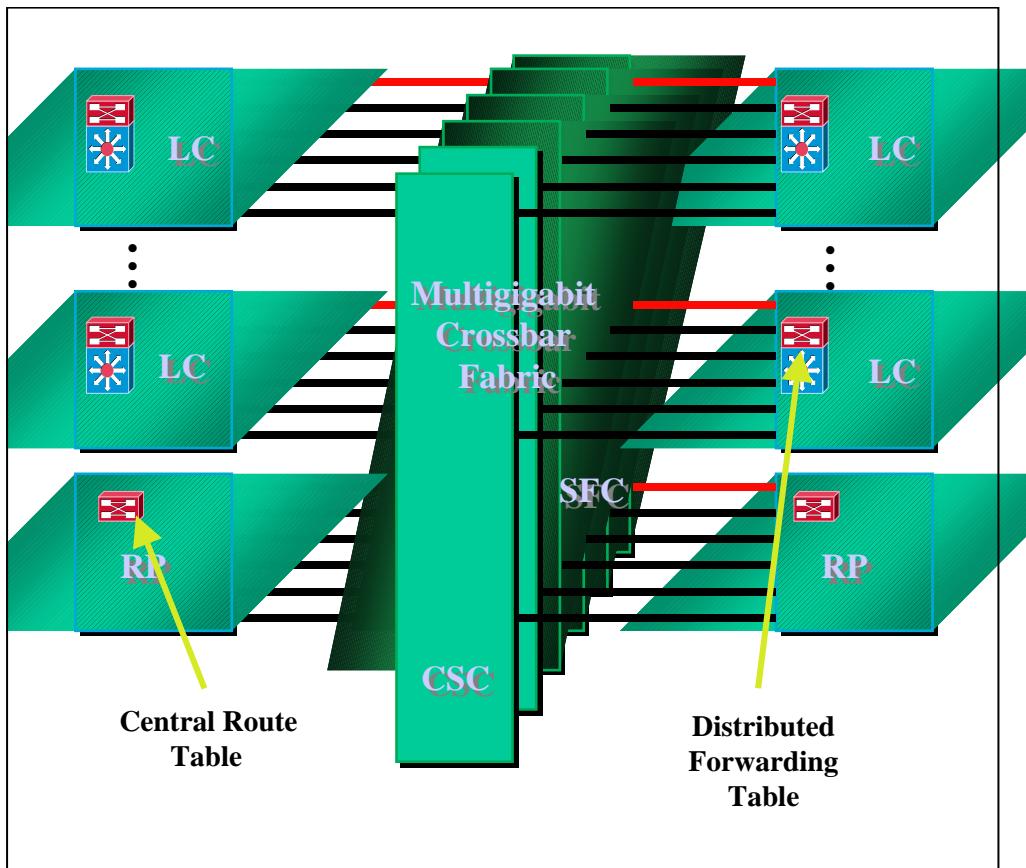
23.1.3.6 Gigabit Switch Router (GSR) Switching Path

The GSR, as shown in Figure 23-23, has a distinct difference in architecture to those platforms previously described.

The router has a Route Processor (RP), which is responsible for the management of the system, the maintenance of routing tables, and the forwarding information base (FIB). Under certain circumstances, the RP will forward packets but these are usually those destined for the RP itself. The GSR is based on the implementation of dCEF in hardware, with the RP copying and maintaining the FIB tables on each of the line cards. Each line card has sufficient CPU processing power to make routing decisions without requiring intervention from the RP.

Unlike the 7500 platform, there is no support for running alternative switching mechanisms. All GSR line cards have hardware dedicated to running dCEF, and all share the same components for a majority of the packet path. Specific ASICs are used for packet processing and, as with the VIP cards, the GSR line cards have their own CPU for FIB and adjacency table support (and in some cases packet processing).

Figure 23-23 GSR Architecture



GSR packet path is divided into three stages:

- Input
- Crossing the switch fabric
- Output

When a packet arrives at the line card interface, it is copied into a buffer. The buffer is taken from the line card's SDRAM and sized appropriately for the received packet. If a buffer is not available, the packet is dropped and the "ignore" counter is incremented on the receiving interface.

Once the packet has been buffered, the switching engine processor decides not only which output line card to forward the packet to but also which port on that line card. The switch engine also makes a decision about which CEF adjacency layer-2 header to use, although the actual layer-2 encapsulation is done on the outbound interface card.

The packet is then enqueued to the switching fabric and sent to the destination line card. Once received by the destination line card, the packet is placed into a buffer. The correct layer-2 header is rewritten into the packet and it is queued to the outbound interface for transmission onto the physical media.

23.1.3.7 Distributed Switching Compatibility Matrix

The various combinations of global configuration switching parameters and interface configuration parameters can be confusing, especially when mixing both distributed Fast Switching and dCEF on the same platform.

The following table tries to clarify the various configuration options available for platforms *apart* from the GSR 120xx.

23.1.3.7.1 With CEF Globally Disabled

CEF can be globally disabled via the global configuration command **no ip cef**. See Table 23-2.

Table 23-2 **Disabled CEF Matrix**

IOS Version	Interface Configuration Command Entered			
	no ip route-cache cef no ip route-cache dist	no ip route-cache cef ip route-cache dist	ip route-cache cef no ip route-cache dist	ip route-cache cef ip route-cache dist
11.1CC	Fast Switched or Optimum Switched	Fast Switched or Optimum Switched	Fast Switched or Optimum Switched	Fast Switched or Distributed Switching
12.0	Fast Switched or Optimum Switched	Fast Switched or Optimum Switched but NOT Distributed Switching	Fast Switched or Optimum Switched	Fast Switched or Optimum switched but NOT Distributed Switching

23.1.3.7.2 With CEF Globally Enabled

CEF is globally enabled by default. However, if previously disabled, it can be reenabled with the global configuration command **ip cef**. See Table 23-3.

Table 23-3 **Enabled CEF Matrix**

IOS Version	Interface Configuration Command Entered			
	no ip route-cache cef no ip route-cache dist	no ip route-cache cef ip route-cache dist	ip route-cache cef no ip route-cache dist	ip route-cache cef ip route-cache dist
11.1CC	Fast Switched or Optimum Switched	Fast Switched, Optimum Switched or Distributed Switching	CEF Switched	Not Applicable
12.0	Fast Switched or Optimum Switched	Fast Switched or Optimum switched but NOT Distributed Switching	CEF Switched	Not Applicable

23.1.3.7.3 With dCEF Globally Enabled

dCEF can be globally enabled using the global configuration command **ip cef distributed**. See Table 23-4.

Table 23-4 dCEF Enabled Matrix

IOS Version	Interface Configuration Commands Entered			
	no ip route-cache cef no ip route-cache dist	no ip route-cache cef ip route-cache dist	ip route-cache cef no ip route-cache dist	ip route-cache cef ip route-cache dist
11.1CC	Fast Switched or Optimum Switched	Fast Switched, Optimum Switched, or Distributed Switching	CEF Switched	Distributed CEF Switched
12.0	Fast Switched or Optimum Switched	Fast Switched or Optimum switched but NOT Distributed Switching	CEF Switched	Distributed CEF Switched

23.1.4 Load Balancing and Other Features

All the previously described switching paths support load balancing across parallel lines in one form or another.

23.1.4.1 Load Balancing along Switching Paths

Process Switching

Process Switching on all platforms supports *per-packet balancing*. Here, packets are transmitted along parallel lines in a round robin fashion. However, there are two inherent drawbacks to this method.

- Process Switching can be *very* CPU-intensive and, therefore, cannot be recommended.
- Balancing on a per-packet basis can lead to packets arriving out of order at the destination host.

Fast Switching, Optimum Switching, Etc.

Fast Switching, along with all other demand-based cache mechanisms, supports *per-destination load balancing* along parallel lines. Here, balancing is achieved on a per-destination prefix basis, where traffic is balanced depending upon the destination network prefix.

Although this method of load balancing does not incur the same level of CPU overhead as per-packet load balancing, and does maintain packet order, it can have the drawback of not distributing the traffic among parallel paths in an equal fashion.

This is because the balancing decision is based on the destination address prefix and, therefore, if there is a significant amount of traffic destined for a single network address, it is likely that all traffic for that destination range will follow the same path.

Evidence of this kind of problem can be seen if one path is predominantly used to the exclusion of any other parallel lines.

CEF and dCEF Switching

Cisco's latest switching mechanisms, CEF and dCEF, allow for both per-packet and per-destination load balancing.

The two main components used by CEF are the *Forwarding Information Base (FIB)* and *Adjacency Table*. The FIB represents a mirror image of the routing table. Any routing topology changes that occur in the routing table are also reflected in the FIB. Because of this, CEF avoids the CPU-intensive tasks associated with demand-based cache management. Also, because the FIB represents the routing table, any equal cost paths present in the table will also be present in the FIB. Each entry in the FIB points to a corresponding entry in the Adjacency Table.

The Adjacency Table maintains information pertaining to all nodes that the router is one layer-2 hop away from. It also contains precomputed layer-2 header information for each adjacency which is used for rewriting layer-2 headers when doing packet switching.

For per-destination load balancing, CEF computes a hash from the source and destination IP addresses. This hash will always point to the same adjacency entry in the adjacency table, ensuring that traffic between the same two session pairs will always follow the same path. Per-destination load balancing is enabled by default.

For per-packet load balancing, the implementation is much simpler because packets are simply distributed over all available paths in a round robin fashion. *It should be noted that the risk of packets arriving out of order at the destination still exists when doing CEF per-packet load balancing.*

Per-packet load balancing is enabled under interface configuration mode with the command **ip load-sharing per-packet**.

Note On the Gigabit Switch Router (GSR), dCEF is enabled by default and cannot be disabled.

Unlike earlier demand-based caching schemes, per-packet forwarding with CEF does not require a drop down into Process Switching. This, coupled with a more scalable architecture, makes CEF an ideal implementation in backbone and core networks.

23.1.4.2 CEF Enabled, Additional Features

Enabling CEF (or dCEF) not only provides the benefits previously mentioned but also enables additional functionality to be implemented:

- Traffic Policing using Committed Access Rate (CAR)
- Unicast Reverse Path Forwarding (RPF)

23.1.4.2.1 CAR

Committed Access Rate (CAR) encompasses a rate-limiting feature that manages a network's access bandwidth policy by ensuring that traffic falling within specified rate parameters is transmitted, while either dropping packets that exceed the predefined rate or transmitting them with a different priority. CAR's default exceed action is to drop packets.

The rate-limiting function of CAR does the following:

- Allows the control of the maximum rate of traffic transmitted or received on an interface.
- Delivers the ability to define layer-3 aggregate or granular incoming/outgoing bandwidth limits.

Additionally, traffic handling policies can be defined for when the traffic either conforms to or exceeds the specified rate limits.

CAR is often configured on interfaces at the edge of a network to limit traffic into or out of the network. The following link contains additional information on CAR:

http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgcr/qos_c/qcpart4/qcpolts.htm#xtocid74402

23.1.4.2.2 Unicast RPF

Unicast Reverse Path Forwarding (RPF), when enabled, checks the source IP address in received packets to ensure that the route back to the source uses the same interface. If the route back to the source does not match the input interface, the packet is discarded. RPF is compatible with both per-packet and per-destination load sharing.

RPF is typically configured at the edge of the network and, when deployed properly, will prevent IP address spoofing, which is the basis for the SMURF Denial of Service attack.

RPF is configured using the following command in interface configuration mode:

ip verify unicast reverse-path

It is important to note that RPF should only be implemented in environments utilizing symmetrical routing since, if a packet's source network is not known out of the receiving interface, the packet will be dropped.

23.1.4.3 Access Lists

Access lists impose an overhead in the processing of a packet since access list entries must be parsed in sequence. As a result, a long or complex access list can impose a significant degradation in packet switching capacity. It was this problem that led to the development of additional mechanisms to help alleviate performance overhead.

Fast Switching/Optimum Switching

As of Release 10.0 of the IOS software, the processing of access lists was incorporated into the Fast Switching code. If the *logging* function is applied, in some circumstances the packet is forced into the Process Switching path. Otherwise, it continues within Fast Switching. If the destination is not known to the fast cache, the packet must be process switched. The result is used to populate the cache as per usual.

The Fast Switching mechanism also applies to NetBIOS access lists. IPX access lists are fast switched but not SSE-switched.

NetFlow Switching

This mechanism was discussed previously. Please see "23.1.3.2.3 NetFlow Switching."

Distributed Switching

When Distributed Switching is enabled, not only are cache entries copied to the line cards but access lists are also held in VIP DRAM. This means that the VIP CPU does not need to refer to the RSP in order to parse a packet against an access list.

Note Named access lists are not currently supported with VIP-based distributed switching.

CEF Switching

In a similar manner to Distributed Switching, copies of access lists, policy-based routing data, and rate limiting rules are held on the line card (both in the GSR and the 75xx with VIPs). A packet is evaluated when it arrives on an interface. If no access lists or other packet-affecting features are being applied to the interface, then the packet is placed into the CEF *Fast-Path*, processed as normal, and forwarded.

If an access list is in effect, then the packets are put into the CEF *Feature-Path*. Each packet must then be parsed against the access list. Therefore, the same effects can be observed: a long or complex access list can impose a significant degradation in packet switching capacity.

If the configuration specifies that a complex access list be used, NetFlow Switching can be enabled in conjunction with CEF to help avoid some of the forwarding overhead. With NetFlow enabled, the first packet of a flow is put into the CEF Feature-Path and a flow tag is created as a result. Subsequent packets are then evaluated against the flow tag as they enter the buffers. If a packet matches the flow tag, it is placed into the CEF Fast-Path.

As noted previously, it is possible that with simple or short access lists, NetFlow Switching in conjunction with CEF may actually lead to a degradation in performance.

23.2 Writing Fast Switching Code

Note This text is outdated, contact fc-uk@cisco.com for the updated CSSR (CEF Scalability and Selective Rewrite) information.

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.

23.2.1 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

23.2.1.1 MCI/CiscoBus Architecture

Note The CSC and Multibus products are not supported in late versions of the Cisco IOS software.

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 double-check 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 actually an 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.

```
charlong sniff1, 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: Introcard

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->writellong = path->vinesp_mh.mac_longs[0];
inreg->writellong = path->vinesp_mh.mac_longs[1];
inreg->writellong = path->vinesp_mh.mac_longs[2];
inreg->writelshort = TYPE_VINES;
inreg->writellong = input->checksum_length;
inreg->writelshort = 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 = (SNAPS_NAP << 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);

```

23.2.1.2 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.

23.2.1.2.1 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);

```

23.2.1.2.2 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.

23.2.1.2.3 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_long;
PUTLONG(macptr++, *cache_macptr++);
PUTLONG(macptr++, *cache_macptr++);
PUTLONG(macptr++, *cache_macptr);
PUTSHORT((ushort *)macptr, TYPE_VINES);

/*
 * Now send the packet.
 */
(*path->idb->hwptra->fastsend) (path->idb->hwptra, pak);
return (TRUE);

```

23.2.2 Software Architecture

There are two types of software architecture for fast switching:

- Full Matrix
- Unique Routines

23.2.2.1 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 $2n$ 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 $2n$ 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 ...
```

23.2.2.2 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($2n$) 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.

23.3 Adding a CEF Feature

Note This text is outdated, contact fc-uk@cisco.com for the updated CSSR (CEF Scalability and Selective Rewrite), and separate new XDR information.

Cisco Express Forwarding (CEF) is a high-performance switching method, comparable in speed to fast switching, with the advantage of being scalable. CEF uses a forwarding information base (FIB) and an adjacency table to make forwarding decisions.

CEF architecture and operation was summarized in earlier sections of this chapter: 23.1.3.2.4 “Cisco Express Forwarding (CEF),” 23.1.4.2 “CEF Enabled, Additional Features,” 23.1.4.1 “Load Balancing along Switching Paths,” and 23.3.4 “Distributed CEF (dCEF).” This section describes CEF from a programming point of view. It includes “Example Forwarding Code” and “Performance Dos and Don’ts.” At the end, it provides a list of “Resources.”

Note Early in CEF’s history, the technology was known by its generic name, FIB. For this reason, the majority of CEF code is located in the directory /vob/ios/sys/ipfib. In the rest of this section, the terms CEF and FIB are used interchangeably.

Reminder: CEF only forwards IP packets. (IPv6 is coming in 2001.)

23.3.1 FIB IDBs

Like the hardware IDB (hwidb) and software IDB (swidb) structures in the larger Cisco IOS software, there are fibhwidb and fibswidb structures within the CEF code. These can be found in /vob/ios/sys/ipfib/ipfib.h.

The fibhwidb and fibswidb structures are essentially copies of the relevant IOS IDBs, with just the IP information needed for CEF. Their layout is carefully crafted to take advantage of cache line fills. Therefore, when adding new fields, think carefully about the correct position for your new field.

23.3.1.1 FIB IDB Subblocks

Like IOS IDBs, the FIB IDBs have grown as new features have been added, resulting in FIB subblocks being designed.

Note Always use FIB subblocks when adding a new feature. See section 23.4, “FIB Subblocks,” below for detailed instructions. Here is an excerpt from that section: “With the final phase of the FIB subblock facility deployed, it is now the FIB team’s goal to stop all field additions to the `fibidbtype` and `fibhwidbtype` data structures, unless it can be truly justified on performance grounds (and believe me—that will be very hard). Additional work will be undertaken to remove the fields that managed to sneak in in the meantime...you have been warned!”

23.3.2 How FIB Technology Works

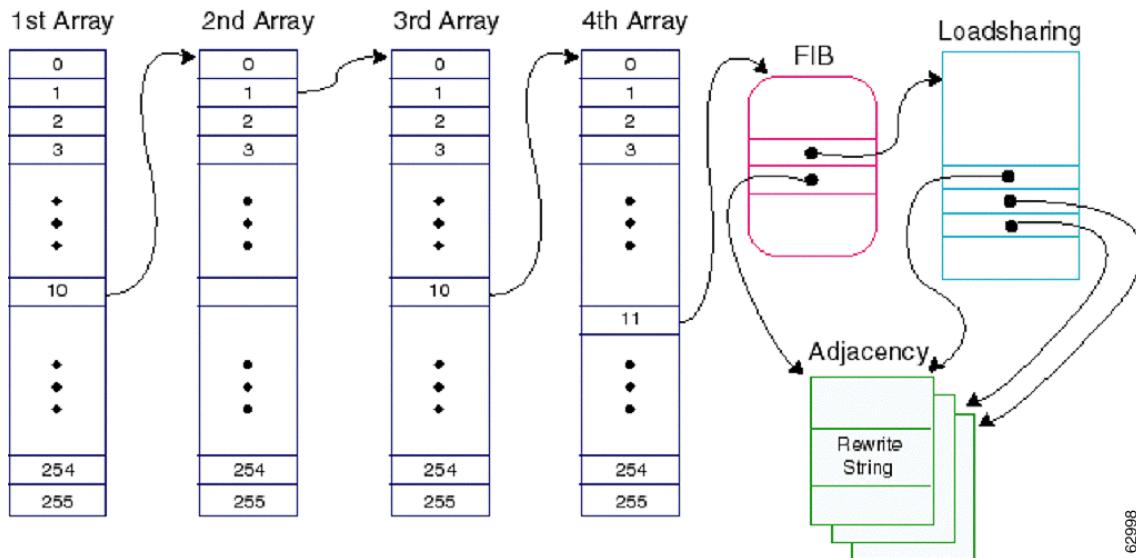
CEF uses the forwarding information base (FIB) to make IP-destination, prefix-based switching decisions. The FIB is conceptually similar to a routing table or information base. It maintains a mirror image of the forwarding information contained in the IP routing table. When routing or topology changes occur in the network, the IP routing table is updated and those changes are reflected in the FIB. This is achieved by the routing code sending an `ndb` to FIB. The NDB contains a number of RDBs, which are individual routing update records.

The FIB maintains next-hop address information based on the information in the IP routing table. Because there is a one-to-one correlation between FIB entries and IP routing table entries, the FIB contains all known routes and eliminates the need for route cache maintenance associated with earlier switching paths, such as fast switching and optimum switching.

As shown in Figure 23-24, FIB tables are stored as four 256-way mtrie tables.

Note An *mtrie* is short for Multi-array reTRIEval tree, the central data structure for enabling a fast and accurate match. A *trie* is a tree structure that uses components of the *key* in order to retrieve entries stored in the structure. The length of the key’s components is known as the *stride*. In the case of CEF, the key is the destination IP address: 8-8-8-8-, as in the example in Figure 23-24.

The GSR (12000 series router) is an exception in that the stride for the mtrie is defined as 16-8-8 rather than 8-8-8-8. The first two key components (8-8-) mtries are combined into a single key component (16-), resulting in a 65536 entry array. This allows the GSR to obtain the adjacency with one less read and, therefore, with higher performance at the cost of more memory for a typically populated trie. (For a sparsely populated table, 8-8- could result in more memory being used than in the 16- case, but this is not a practical example.)

Figure 23-24 FIB Tables

62998

Figure 23-24 shows the lookups for the adjacency of IP address 10.1.10.11. The most significant 8 bits of an address are the index into the first array. At this location, a pointer is retrieved that points at the relevant array. Although Fig 22-24 shows only one 2nd Array, in effect there are as many as 256 of these. The pointer in the first array indicates which of these second arrays to pick. So the first array gives you the second array, and so on until a leaf turns up. This leaf represents the longest match for the IP address driving the lookup and points to the FIB block. In the case of a prefix for a network, the FIB can be reached with fewer lookups.

The FIB structure retrieved at the end of the tree traversal is the leaf (block), which contains a pointer to the adjacency. The adjacency has the link-layer header, counters, various flags connected with the address/prefix, and other information. If loadsharing has been disabled, the FIB block contains a pointer directly to the adjacency. Otherwise, the multiple adjacencies are pointed to by the loadsharing array.

It is important to remember that FIB switching is quite different from cache-based fast switching. This means that, as feature writers, you should not depend on various tricks like invalidating sections of the cache in order to get packets to be process switched.

On some of the high-end platforms, for example the 7200 and 7500, a *fake static pak* is available, which is not fully populated. It should be noted that any changes made to this fake pak will not be available to other switching paths in the event of the packet being punted up. (For a discussion of fast switching, see “Fast Switching” in *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.)

23.3.2.1 Adjacency Tables

As mentioned, locating the structure found at the leaf (`fibtype`) gives you a pointer to the correct entry in the *adjacency table*. The adjacency table is populated as adjacencies are discovered. Each time an adjacency entry is created, such as through the ARP protocol, a link-layer header for that adjacent node is precomputed and stored in the adjacency table. It is usually called a *rewrite string*, as in Figure 23-24.

The adjacency (not rewrite string) contains the output queue information, the prepend Layer-2 addressing information, output MTU, and other information. This structure can be found in `/sys/routing/adjacency.h`.

In the case of load balancing, the entry in `fibtype` contains an array of paths leading to the prefix that it pertains to, each path describing which next hop should be used for the path. The information describing this next hop includes a pointer to the adjacency leading to that next hop. That is, it includes a pointer to a number of different adjacencies. These are referred to as *paths* in FIB technology.

The selection of which path to use depends on the configuration. In the simple case of per-packet load balancing, each path is used on a round-robin basis.

If the returned adjacency MTU is smaller than the packet, the packet used to be punted up to the process switching code. This is no longer true. Fragmentation happens in CEF under interrupt in recent releases. It is reassembly that still forces process switching. Refer to `pas_ipfib_fragment_pkt()` in recent releases.

23.3.2.1.1 Special Adjacency Types

There are several adjacency types for exception processing.

- Null adjacency

Packets destined for a Null0 interface are dropped. This can be used as an effective form of access filtering, without the need to implement ACLs (access control lists).

- Glean adjacency

When a router is connected to a point-to-multipoint interface such as Ethernet (directly to several hosts), CEF maintains a prefix for the network in the FIB table, but does not initially have specific host entries for each host on the network. (The FIB table on the router maintains a prefix for the whole subnet.) This subnet prefix points to a *glean adjacency*. When packets need to be forwarded to a specific host, the FIB lookup results in a glean adjacency. This packet is punted up to the process level for ARP resolution. When the ARP response is received, and the ARP table is populated, this also drives the creation of a host prefix in the FIB table. This prefix is known as an `adjfib` and links the host prefix to a normal adjacency, which will result in packets to that prefix being CEF switched as normal in the future. (The specific IP address is added to the FIB and the adjacency table.)

- Punt adjacency

Features that require special handling or are not yet supported within CEF switching paths are forwarded to the next switching layer for handling.

Note ARP resolution can only occur at process switching level (non-interrupt). However forwarding can occur at the next higher switching level if the feature/encapsulation is not supported at the lowest level. That is, dCEF ->CEF->fast switching->route processing can all try to forward the packet.

A punt adjacency could not be installed on GSR routers (Cisco 12000 series). This is because there is no other (lower-performance) packet switching layer in the GSR.

- Discard adjacency

Packets are discarded. This type of adjacency occurs only on the GSR routers (12000 series).

- Drop adjacency

Packets are dropped, but the prefix is checked. This is to allow for extra accounting.

23.3.2.2 Example Forwarding Code

The following code fragment is the feature path, meaning the path that the packet would take if some features have been enabled. It was taken from a FIB-based LES feature's fastswitching routine, `les_ipfib_feature_switch()`.

```

/*
 * Do FIB lookup and switch. If FIB lookup failed, check if
 * the packet should be punted to next level or dropped.
 */

destination = GETLONG(&ip->dstdadr);
fib = ipfib_find_fib(destination);

if (!fib) {
    /*
     * Check if it's ok to drop this packet after FIB lookup failure. If it
     * is ok, drop the packet otherwise instruct the caller to punt the
     * packet.
     */
    fail = ipfib_noroute_dropcheck(input, swinput);
    return (return (les_ipfib_failed_wrapper(pak, ip, input, NULL,
                                              destination, fail, ip_feature_fastswitch)));
}

adj = ipfib_find_adj(ip, destination, input, fib, &fail, &tag_rew);
if (!adj) {
    /*
     * Check if it is ok to drop this packet after adjacency lookup failure.
     * If it is ok, drop the packet otherwise instruct the caller to punt the
     * packet.
     */
    return (les_ipfib_failed(pak, ip, input, NULL, destination, fail,
                           ip_feature_fastswitch));
}

/*
 * As long as no fragmentation is required by output adjacency MTU switch the
 * packet; otherwise punt up
 */

if (les_ipfib_switch_pkt(pak, ip, input, input_size, adj, TRUE) == IPFLOWOK)
    return (TRUE);
else
    return (ip_feature_fastswitch(pak));

```

23.3.2.3 Background Processes

There are a number of running background processes that are called from `fib_background`:

- `ip_fib_slow_processing`

Called every `FIB_SLOW_TIMER` (15 seconds). Used to resolve the queue of recursive and unresolved routes.

- `ip_fib_age_arp_throttle_element`

Called every `FIB_ARP_THROTTLE_TIMER` (1 second). Used to throttle the rate of sending ARP requests for a given destination address to 1 per second.

- `adjacency_update_hwidb_stats`

Called every FIB_STAT_UPDATE_TIMER (1 second). Used to update the hwidb with the counters from the fibhwidb. The fibhwidb counters are zeroed.

- `ip_fib_compute_load`

Called every FIB_COMPUTE_LOAD_TIMER (5 Seconds). Used to calculate the 5-minute exponentially-decayed output bit rate.

When the FIB table is being downloaded/repopulated in the line cards, if a FIB delete messages occurs, the `fib_delete_timer` is started (FIB_DELETE_TIMER=5 seconds). All the deleted messages are queued and processed by `ipfib_purge_defered_delete` when the timer expires.

There is also a watched process called `fib_scanner`. This performs various FIB maintenance tasks such as checking whether the loadsharing information is still valid and deallocating it if needed (`ipfib_check_loadsharing`).

`fib_check_adjacency` ensures that all paths in the FIB entries point to a valid adjacency.

23.3.3 Guidelines for Adding a New Feature

In `ipfib/iphfifeature.h`, add code to either `ipfib_check_early_features()` for input-based features, or `ipfib_check_features()` for output-based features:

```
static inline int
ipfib_check_features (paktype *pak, iphdrtype *ip, idbtype *swinput,
                      hwidbtype *input, fibhwidbtype **output_ptr,
                      fibtype *fib, adjacency **adj_ptr, void **tag_rew_ptr,
                      ipflow_t *flow, unsigned input_flags,
                      fibidbtype *input_fibidb, idbtype *cfg_swinput,
                      boolean full_pak, boolean distributed)

static inline int
ipfib_check_early_features (paktype *pak, iphdrtype *ip, idbtype *swinput,
                           hwidbtype *input, ipflow_t *flow,
                           fibtype **fib_ptr, adjacency **adj_ptr,
                           void **tag_ptr, unsigned input_flags,
                           fibidbtype *input_fibidb, idbtype *cfg_swinput,
                           boolean full_pak, boolean distributed,
                           boolean use_fib_acl)
```

An IP fast flag will need to be allocated for the feature. Give thought to the proper place in the order for your feature.

Keep to a minimum the code added to the feature path. Check the flags and perform a `reg_invoke` into the feature code.

If the feature removes the packet from the switching path (`IPFLOWERR_CONSUME`) or changes the output interface, insure that it does not skip any critical features that might follow. For example, check the output access lists of the original output interface.

Changing the actual packet in packet memory is strongly discouraged. This causes problems for later features and also in the event that a later feature needs to be punted.

When referencing packet memory, code located in the “turbo” path must not reference beyond the TCP header in the packet. This would result in data corruption bugs and other strange problems surfacing in seemingly unrelated code. This is due to the fact that, on the RSP, the “turbo” path only invalidates the first two cache lines in the CPU’s data cache. See `src-rsp/rsp_if.h` for details on how the packets are aligned with respect to cache line boundaries.

If a feature is added to any path, also support all the paths to which it could punt. This avoids cases where the only path that supports a certain combination of features is process switching.

Note One more reminder: all new features should use *FIB subblocks* to store data.

23.3.3.1 Performance Dos and Don'ts

Be conscious of the data cache on many platforms. Maintain good cache locality in data structures referenced in the fast path. Do not add new fields to other data structures without considering their position. Understand that packet memory is cached on some platforms (7500) and not on others (7200, 3600). Understand that the RSP1/2 does not have any L2 cache and that space in the L1 cache is tight.

If a new feature is not implemented in all paths, it is still often possible to identify packets that would be affected by the feature and punt only those packets to the appropriate path. For features that do not affect all packets, having identified the affected packets will greatly improve the aggregate box performance.

NetFlow Feature Acceleration (ENG-22487) can be used to reduce the overhead in identifying packets of interest to a particular feature. From a performance standpoint it is good to determine whether a particular feature gets enough savings to make up for the addition of netflow overhead. In situations where there are multiple features or where netflow is desirable for accounting purposes, the performance benefit is more obvious. In any case, packet-by-packet forwarding code for the feature is required; do not assume netflow acceleration will always be available.

23.3.3.2 Design, Code Reviews, and Questions

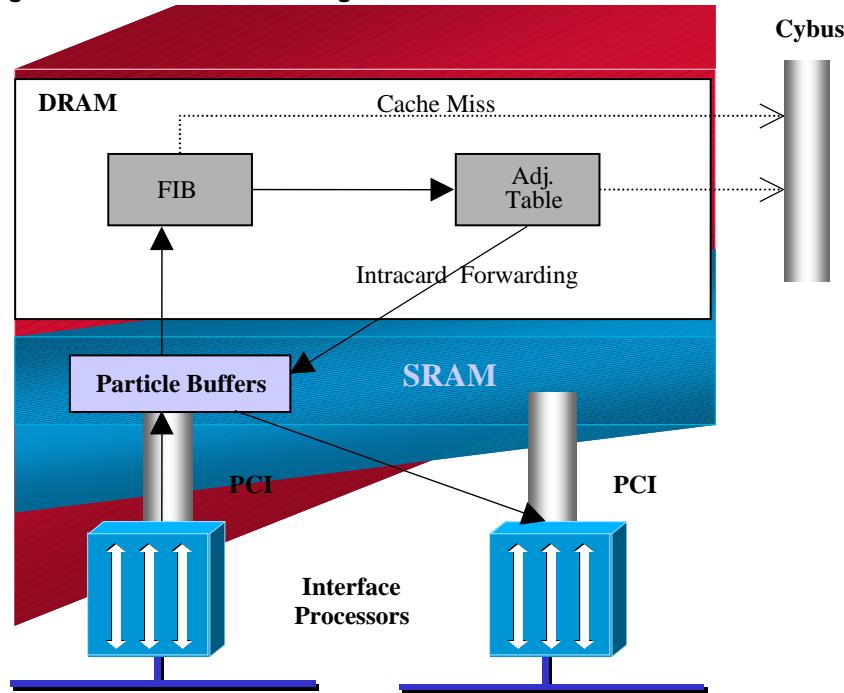
Having feature design comments arise in the code review process is frustrating for reviewer and developer alike. Please avoid this eventuality by sending a design review past the alias `interest-fib` early in the coding process. This should help make the code review by the same alias more pleasant. Also direct any questions to the same alias.

23.3.4 Distributed CEF (dCEF)

To improve the scalability of high-end routers, the CEF tables can be distributed to special intelligent line cards such as VIP line cards (7500) or Gigabit Switch Router (GSR) line cards. These each maintain an identical copy of the FIB and adjacency tables. On the GSR the FIB tables are used to program switching hardware. The FIB and adjacency databases are transported via interprocess communication (IPC) to the line cards and packets are switched using those replicated databases. The IPC mechanism ensures synchronization of FIBs and adjacency tables on the route processor and line cards. These are called *XDR blocks*. XDR is a FIB IPC overlay mechanism. Only a “fake” pak is available to feature writers and changes to it are not propagated to other switching modes. Also, any data structures used by a feature must be transported to the line card via FIB IPC and the statistics sent back to the RP.

Figure 23-25 shows the relationship between the route processor and line cards when dCEF mode is active.

Figure 23-25 dCEF Switching



In this GSR (12000 series) router, the line cards perform the switching. In other routers, where you can mix various types of cards in the same router, it is possible that not all of the cards you are using support CEF. When a line card that does not support CEF receives a packet, the line card forwards the packet to the next higher switching layer (the route processor). This structure allows legacy interface processors to exist in the router with newer interface processors.

23.3.4.1 Load Balancing for CEF

CEF load balancing is based on a combination of source and destination packet information. It allows you to optimize resources by distributing traffic over multiple paths for transferring data to a destination. You can configure load balancing on a per-destination or per-packet basis. Load balancing results in extra information being stored in the FIB structure. These are referred to as *paths*. Currently, up to a maximum of six paths are supported. Within each path structure there is a pointer to an adjacency.

23.3.4.1.1 Per-Destination Load Balancing

Per-destination load balancing allows the router to use multiple paths to achieve load sharing. Packets for a given source-destination host pair are guaranteed to take the same path, even if multiple paths are available. This guarantee is achieved by performing a hash function on the source and destination IP addresses, which is then masked down to a number in the range of 0-15. This gives you a path *bucket number*.

For equal distribution of load, each adjacency appears the same number of times in the path's array. For unequal distribution, the adjacencies are distributed over the whole array in the approximately correct proportions. If the proportions do not allow the whole array to be used, then a *maximum bucket used* figure is used, to limit the hash function range.

Bucket Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---------------	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

Equal with 4 paths a b c d a b c d a b c d a b c d

a wants 2*b

b wants 2*c a b a c a b a d a b a c a b a ~

c wants 2*d

The result achieved then is that traffic destined for different IP pairs tends to take different paths. Per-destination load balancing is enabled by default when you enable CEF and is the load balancing method of choice for most situations.

23.3.4.1.2 Per-Packet Load Balancing

Per-packet load balancing allows the router to send successive data packets over paths without regard to individual hosts or user sessions. It uses the round-robin method to determine which path each packet takes to the destination. Per-packet load balancing ensures balancing over multiple links.

While path utilization with per-packet load balancing is good, packets for a given source-destination host pair might take different paths. Per-packet load balancing could introduce reordering of packets. This type of load balancing would be inappropriate for certain types of data traffic that depend on packets arriving at the destination in sequence, such as voice traffic over IP.

Use per-packet load balancing to help ensure that a path for a single source-destination pair does not get overloaded. If the bulk of the data passing through parallel links is for a single pair, per-destination load balancing will overload a single link while other links have very little traffic.

Enabling per-packet load balancing allows you to use alternate paths to the same busy destination.

To enable per-packet load balancing, perform the following task in interface configuration mode:

```
(config-if)# ip load-sharing per-packet
```

Note Per-packet load-balancing is now supported on the GSR platform.

23.3.4.2 Debugging

This section discusses CEF debugging commands and shows example output of some useful ones.

23.3.4.2.1 Show Commands

- **show tech-support cef**

This command is very useful. It shows all the normal tech-support **show** commands and lots of CEF-related information as well. It can also be executed on the VIP line cards.

- **show ip cef internal**

Use this command to see the internal CEF table. This command produces a lot of output! This example output has been edited:

```
ubr2# show ip cef internal
IP CEF with switching (Table Version 37134)
 37069 routes, 0 reresolve, 0 unresolved (0 old, 0 new)
 37069 leaves, 3838 nodes, 8439800 bytes, 37319 inserts, 250 invalidations
 0 load sharing elements, 0 bytes, 0 references
 2 CEF resets, 0 revisions of existing leaves
 0 in-place modifications
 refcounts: 1053083 leaf, 982784 node

Adjacency Table has 965 adjacencies
 5 incomplete adjacencies

 0.0.0.0/0, version 193, cached adjacency 24.124.5.254 Default route
 0 packets, 0 bytes
   via 24.124.5.254, 0 dependencies, recursive
    next hop 24.124.5.254, FastEthernet0/0 via 24.124.5.254/32
     valid cached adjacency
 6.0.0.0/8, version 569, cached adjacency 24.124.5.254 Recursive route
 0 packets, 0 bytes
   via 144.228.115.213, 0 dependencies, recursive
    next hop 24.124.5.254, FastEthernet0/0 via 144.228.115.212/30
     valid cached adjacency
 24.124.5.0/32, version 4, receive
 24.124.5.1/32, version 188, connected, cached adjacency 24.124.5.1 Reachable
node on subnet
 0 packets, 0 bytes
   via 24.124.5.1, FastEthernet0/0, 0 dependencies
    next hop 24.124.5.1, FastEthernet0/0
     valid cached adjacency
 24.124.5.249/32, version 36978, connected, cached adjacency 24.124.5.249
Reachable node on subnet
 0 packets, 0 bytes
   via 24.124.5.249, FastEthernet0/0, 1 dependency
    next hop 24.124.5.249, FastEthernet0/0
     valid cached adjacency
 24.124.5.252/32, version 3, receive Our address on subnet
 24.124.5.255/32, version 5, receive Broadcast address on subnet
```

23.3.4.2.2 Debug Commands

There is also a list of common debug commands.

```
ed6-75a# debug ip cef ?
accounting      Accounting events
drops          Packets dropped by CEF
events         IP CEF table events
hash           IP CEF hash events
interface-ipc Interface events related IPC
ipc            IP CEF IPC events
prefix-ipc    IP-prefixes related IPC
receive        Packets received by IP CEF
subblock       IP CEF subblock events
table          IP CEF table changes
```

Example output:

```
Nov 13 12:31:21.075: CEF-IP: Receive address 99.0.0.0/32 already exists
Nov 13 12:31:21.075: CEF-IP: Receive address 99.255.255.255/32 already exists
Nov 13 12:31:21.075: CEF-Table: Event up, 2.0.0.0/8
Nov 13 12:31:21.075: CEF-Table: Event up, 197.12.11.0/24
Nov 13 12:31:21.087: CEF-Table: Event up, 197.21.251.0/24
Nov 13 12:31:21.087: CEF-Table: Event up, 192.18.2.0/24
Nov 13 12:31:21.111: CEF-IPC: ipc sent. Slot 1 Seq 0
Nov 13 12:31:21.111: CEF-IPC: ipc sent. Slot 1 Seq 0
Nov 13 12:31:21.111: CEF-IPC: ipc sent. Slot 4 Seq 0

Nov 13 12:31:36.067: CEF-Table: attempting to resolve 197.12.12.0/24
Nov 13 12:31:36.067: CEF-IP: resolved 197.12.12.0/24 via 99.0.0.1 to 99.0.0.1
Loopback1
Nov 13 12:31:36.067: CEF-Table: attempting to resolve 197.12.11.0/24
Nov 13 12:31:36.067: CEF-IP: resolved 197.12.11.0/24 via 99.0.0.1 to 99.0.0.1
Loopback1
Nov 13 12:31:57.695: CEF-IP: Checking dependencies of 12.0.0.0/8
Nov 13 12:31:57.695: CEF-Table: Adjacency-prefix 12.0.0.1/32 add request --
succeeded

Nov 13 12:33:03.079: CEF-Table: Flushing entry for 195.27.191.0/24
Nov 13 12:33:03.079: CEF-Table: Flushing entry for 195.31.54.0/24
Nov 13 12:33:03.079: CEF-Table: Flushing entry for 195.181.32.0/24
Nov 13 12:33:03.079: CEF-Table: Flushing entry for 197.12.11.0/24
```

23.3.4.3 Resources

- 1 CEF team's home page:

http://wwwin-eng.cisco.com/Eng/IOS/IOS_Euro/CEF

- 2 CEF nerd lunch by Neil Jarvis:

http://wwwin-eng.cisco.com/Eng/IOS/IOS_Euro/CEF/docs/nerd-lunch/

- 3 FIB switching nerd lunch by Ravi Chandra:

There is a video available from the video library available at:

<http://wwwin-eng.cisco.com/video/video.shtml>

- 4 CEF switching white paper by Martin McNealis:

http://wwwin.cisco.com/cmc/cc/pd/iosw/iore/tech/cef_wp.htm

- 5 CEF presentation by Vijay Bollapragada:

http://www-tac.cisco.com/Support_Library/Internetworking/CEF/Training/presentation/index.htm

- 6 CEF switching by Tom Vo:

<http://wwwin-people.cisco.com/~cmujie/fib+ipservices/cef-n105.pdf>

- 7 CEF commands:

http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgr/switch_r/xrcef.htm

- 8 FAQ on FIB:

http://www-tac.cisco.com/Support_Library/People/lwigley/CEF/FAQ.html

9 FIB subblocks documentation by Neil Jarvis:

Section 23.4 of this chapter, “FIB Subblocks”

10 Email questions to:

interest-fib

23.4 FIB Subblocks

Note This text is outdated, contact fc-uk@cisco.com for the updated CSSR (CEF Scalability and Selective Rewrite) information.

This is Neil Jarvis’s “FIB Subblocks Documentation,” from /vob/ios/sys/Doc/FIB_subblocks. It was incorporated into this chapter in July 2001.

Terms (also see Terms at the beginning of this chapter)

function table

(ft) Static data structure that describes how a feature uses the FIB subblock facility.

line card

(LC) Slave processor, controlled from the RP. Only distributed environments have line cards.

route processor

(RP) Centralized processing engine. In a nondistributed environment, this is the only processor in the system.

23.4.1 Introduction to FIB Subblocks

This section describes the history and architecture of the FIB subblock facility.

23.4.1.1 What Are FIB Subblocks and Why Are They Needed?

A subblock is a block of memory that contains a set of related data that is used to implement a feature in IOS. Subblock data structure allocation is managed by the feature. The subblock is only created when the feature is present and in use. Subblocks are associated with interfaces, both physical and virtual.

Since versions 11.3/12.0 of the Cisco IOS software, subblocks have been linked to the IDB data structures: `idbtype` and `hwidbtype`. This section describes the FIB subblock facility, which allows subblock data to be associated with the FIB data structures: `fibidbtype` and `fibhwidbtype`.

Because the FIB can operate in distributed environments, the FIB subblock facility supports the automatic synchronization of subblock data between the central processor and the distributed processors, something that normal subblocks cannot accomplish.

The FIB subblocks facility was added in an effort to stem the data bloat that was being seen with `fibidbtype` and `fibhwidbtype` structures. As more and more features were being implemented in FIB (a very good thing), more feature-specific data was being added to the `fibidbtype` and `fibhwidbtype` structure (a very bad thing).

With the final phase of the FIB subblock facility deployed (implementation history in 23.4.1.2), the current goal of the FIB team is to stop all field additions to the `fibidbtype` and `fibhwidbtype` data structures unless the addition can be truly justified on performance grounds (and believe me, that will be very hard). Additional work will be undertaken to remove the fields that managed to sneak in in the meantime...you have been warned!

23.4.1.2 Where Is the FIB Subblock Facility Supported?

The FIB subblock facility was implemented in three phases, tracking the requirements of the Unicast RPF feature as it was enhanced. (Unicast Reverse Path Forwarding (RPF) verifies whether the source IP is reachable, to prevent malformed or forged source IP addresses from entering a network.) Phase one ported the normal subblock facility to use the `fibidbtype` and `fibhwidbtype` structures. It also added automatic LC-to-RP (line card to route processor) statistic synchronization. Phase one was committed as CSCdk70183. Phase two added automatic RP-to-LC control synchronization, and was committed as CSCdp76668 (and CSCdp89248 which fixed a problem introduced by the first commit). Phase three added out-of-band subblock data creation and deletion and automatic LC-to-RP event synchronization. Phase three was committed as CSCdr12914.

FIB subblocks, phase three, was committed to 12.1 (delaware) on April 19, 2000.

23.4.1.3 Differences Between FIB Subblocks and Normal Subblocks

Apart from the obvious difference that FIB subblocks support automatic synchronization in a distributed environment, FIB subblocks look identical to the fast access instances of the normal subblocks.

Because FIB is a performance critical piece of software, it was decided that the additional runtime overhead of supporting the normal subblock concept of dynamic FIB subblock types was not justified.

23.4.1.4 Nondistributed and Distributed Environments

When discussing the FIB subblock facility, it must be recognized that any feature using the facility must be able to operate in two environments, *nondistributed* and *distributed*. Examples of the nondistributed environment are the c3640 and c7200. Examples of the distributed environment are the c7500, c12000, and c5850. By structuring your code carefully, you can minimize the amount of duplicated or redundant code.

Obviously the nondistributed environment does not require the synchronization support. However, the current IOS build environment does not understand the concept of a nondistributed FIB build. Instead, it uses the subsystem facility to include/exclude certain files from builds for, say, the c3640 versus the c7500. For non-trivial features, this approach will work fine, allowing three initialization/registry code segments to be employed.

For trivial features, using subsystems to include/exclude FIB subblock feature code is overkill and a compromise is suggested: the file(s) used to implement the RP FIB subblock portion of a feature are also used on nondistributed platforms (a file of this type has the FIB naming convention of `ipfib_ios_xxxx.c`). This means that nondistributed images have unnecessary code but, as you will see from the documentation below, it is a small amount and the overhead is worth the savings in dealing with subsystems.

23.4.1.5 Example FIB Subblock Implementation

If you look in `/vob/ios/sys/templates`, you will find four files that contain a complete but noddy (simple) example of FIB subblocks:

```
/vob/ios/sys/templates/ipfib_xxxxxsb.h
/vob/ios/sys/templates/ipfib_xxxxxsb.c
/vob/ios/sys/templates/ipfib_ios_xxxxxsb.c
/vob/ios/sys/templates/ipfib_lc_xxxxxsb.c
```

Copy the files to the `/vob/ios/sys/ipfib` directory and rename them, replacing the `xxxx` with your feature's identifier, per Table 23-5.

Table 23-5 FIB Subblock .h and .c Files

File	Description
<code>ipfib_xxxxxsb.h</code>	Contains the FIB subblock data block and the <code>ipfib_xxxxxsb.c</code> function prototype definitions
<code>ipfib_ios_xxxxxsb.c</code>	Implements the nondistributed and RP code for the feature, as described in the previous section, “Nondistributed and Distributed Environments”
<code>ipfib_lc_xxxxxsb.c</code>	Contains the line card code

To add these files to the build system, you need to update `/vob/ios/sys/makesubsys`. Add `ipfib_xxxxxsb.o` to `sub_fib_common`, `ipfib_ios_xxxxxsb.o` to `sub_fib_ios`, and `ipfib_lc_xxxxxsb.o` to `sub_fib_lc`.

The examples in this section were cut and pasted from this set of example files, so feel free to go and take a look now....

23.4.1.6 fibidbtype and fibhwidbtype Subblocks

FIB subblocks are supported for both `fibidbtype` and `fibhwidbtype` data structures. The facilities provided for both data structures are identical. Still, because the facility must interact with these data structures at the lowest level, all FIB subblock code and data structures are duplicated: once for `fibidbtype` (look for `fibsbsb` and `fibidb` naming conventions) and once for `fibhwidbtype` (look for `fibsbsb` and `fibhwidb` naming conventions).

To simplify this documentation, all examples and discussions use the FIB subblocks associated with the `fibidbtype` structure.

OK. Enough blurb. How Exactly Do I Use FIB subblocks?

Not so fast, we need to take a little detour to describe the FIB subblock function table.

23.4.1.7 FIB Subblock Function Table

Like the existing subblock facility, FIB subblocks are controlled via function tables. Here is the data type of the function table (for fibidb type subblocks):

```
typedef struct fibswsb_ft_ {
    fibswsb_id_t fibswsb_id;           /* Type identifier */
    ulong fibswsb_flags;              /* Flags */
    fibswsb_bool_t fibswsb_destroy;    /* Remove subblock and free */
    fibswsb_void_t fibswsb_enqueue;    /* Reattach subblock */
    fibswsb_void_t fibswsb_unlink;     /* Detach subblock from system */
    fibswsb_xdr_encode_t fibswsb_xdr_control_encode; /* Encode control */
    fibswsb_xdr_decode_t fibswsb_xdr_control_decode; /* Decode control */
    fibswsb_xdr_size_t fibswsb_xdr_control_size; /* Return size of control */
    fibswsb_xdr_encode_t fibswsb_xdr_stat_encode; /* Encode stats */
    fibswsb_xdr_decode_t fibswsb_xdr_stat_decode; /* Decode stats */
    fibswsb_xdr_size_t fibswsb_xdr_stat_size; /* Return size of stats */
    fibswsb_xdr_encode_t fibswsb_xdr_event_encode; /* Encode event */
    fibswsb_xdr_decode_t fibswsb_xdr_event_decode; /* Decode event */
    fibswsb_clear_counters_t fibswsb_clear_counters; /* Clear counters */
    fibswsb_show_sb_t fibswsb_show_sb; /* Print subblock data - debug */
} fibswsb_ft;
```

Don't be daunted by the large size of this structure. It's not that bad!

- **fibswsb_id**

The first field, `fibswsb_id`, contains the unique identifier for this subblock. Each feature assigns itself a new identifier. This is done in the `ipfib/iphfib_subblock.h` file. For `fibidbtype` subblocks, the enum `fibswsb_id_t` gets a new element (add it before `FIBSWSB_MAX`). For `fibhwidbtype` subblocks, the enum `fibhwsb_id_t` gets a new element (add it before `FIBHWSB_MAX`). In both cases, there is a string array in `ipfib_subblock.c` that needs to get the feature's identification string. If you forget to add this, expect the FIB subblocks debugging messages to report unknown subblock types.

- **fibswsb_flags**

The second field, `fibswsb_flags`, is a bit mask of flags, formed by or-ing together sets of flags:

The first set of flags takes one of three possible values that define the type of synchronization that will be required when the data is pushed down from the RP to the LCs:

- **FIBSB_INPUT_ONLY**: The subblock data is only used on the input path of the FIB code. This will limit the distribution of the subblock data to the line card on which the `fibidb` really resides. This minimizes IPC traffic and memory requirements on the LCs.
- **FIBSB_OUTPUT_ONLY**: The subblock data is only used on the output path of the FIB code. Because this information is required by all line cards (output features are done on the receiving line card, the output interface may be on any line card), the subblock data will be distributed to all line cards.
- **FIBSB_INPUT_AND_OUTPUT**: The subblock data is used on both the input and output paths of the FIB code. Because this information is required by all line cards, the subblock data will be distributed to all line cards.

The second and third set of flags are used to indicate if the corresponding encode/decode routines (control, stat, and event) are interrupt-safe. Because the routines are called from process level, they cannot modify data structures that may also be modified by interrupt-driven code. Updating stats is a good example. If the flag for a routine is set to `FIBSB_xxxx_IRQ_SAFE`, the routine is interrupt-safe and the FIB subblock code does not disable interrupts when it makes the call. If,

on the other hand, the flag is set to `FIBSB xxxx IRQ UNSAFE`, the routine is not interrupt-safe and the FIB subblock code will disable interrupts when it makes the call. There are twelve flag values in total. See `ipfib_subblock.h` for their names.

- `fibswsb_destroy`, `fibswsb_enqueue`, and `fibswsb_unlink`

The next three fields, `fibswsb_destroy`, `fibswsb_enqueue`, and `fibswsb_unlink`, represent subblock access functions and are equivalent to the normal subblock function table. If no special access functions are required (probably the normal case), you can use three generic routines that the FIB subblock facility provides: `fibswsb_delete`, `fibswsb_link`, and `fibswsb_unlink`.

- `fibswsb_xdr_control_encode`, `fibswsb_xdr_control_decode`, and `fibswsb_xdr_control_size`

The next three fields, `fibswsb_xdr_control_encode`, `fibswsb_xdr_control_decode`, and `fibswsb_xdr_control_size`, are used to implement the feature-specific portion of RP-to-LC control data synchronization. These functions are described in section 23.4.3.1.1.

- `fibswsb_xdr_stat_encode`, `fibswsb_xdr_stat_decode`, and `fibswsb_xdr_stat_size`

The next three fields, `fibswsb_xdr_stat_encode`, `fibswsb_xdr_stat_decode`, and `fibswsb_xdr_stat_size`, are used to implement the feature-specific portion of LC-to-RP statistic data synchronization. These functions are described in section 23.4.3.1.2.

- `fibswsb_xdr_event_encode` and `fibswsb_xdr_event_decode`

The next two fields, `fibswsb_xdr_event_encode` and `fibswsb_xdr_event_decode`, are used to implement the feature-specific portion of LC-to-RP event data synchronization. These functions are described in section 23.4.3.1.3.

- `fibswsb_clear_counters`

The next field, `fibswsb_clear_counters`, can be set up with a feature function that will be invoked when the user issues a **clear counters** command. It is only needed in the RP implementation since the LC only sends deltas.

- `fibswsb_show_sb`

The final field, `fibswsb_show_sb`, is a debugging helper function. If present, when the **show cef interface internal** command is issued, each subblock is called via this function to print some debug output. See “Debugging Subblocks,” section 23.4.4, for more details.

23.4.1.8 Example: FIB Function Tables

Whew, glad we’re done with that. So what you are asking now is, “How do I define and use this structure?” Below are two examples of how this structure is set up. Both are used to implement the test feature. The first instance of the function table is used in nondistributed and RP images. This is an example of the overhead that nondistributed environments will inherit: the XDR functions defined in the structure will never be called.

```
const static fibswsb_ft ipfib_testsbtios_ft = {
    FIBSWSB_TEST,
    (FIBSB_INPUT_AND_OUTPUT | FIBSB_CONTROL_ENCODE_IRQ_SAFE |
     FIBSB_STAT_DECODE_IRQ_UNSAFE | FIBSB_EVENT_DECODE_IRQ_SAFE),
    fibswsb_delete,
    fibswsb_link,
    fibswsb_unlink,
    ipfib_testsbtios_xdr_control_encode, NULL, ipfib_testsbtios_xdr_control_size,
    NULL, ipfib_testsbtios_xdr_stat_decode, NULL,
```

```

        NULL, ipfib_testsbt_xdr_event_decode,
        ipfib_testsbt_clear_counters,
        ipfib_testsbt_show_sb
    };
```

On the LC, the function table would be set up like this:

```

const static fibswsb_ft ipfib_testsbt_lc_ft = {
    FIBSWSB_TEST,
    (FIBSB_INPUT_AND_OUTPUT | FIBSB_CONTROL_DECODE_IRQ_SAFE |
     FIBSB_STAT_ENCODE_IRQ_UNSAFE | FIBSB_EVENT_ENCODE_IRQ_UNSAFE),
    fibswsb_delete,
    fibswsb_link,
    fibswsb_unlink,
    NULL, ipfib_testsbt_xdr_control_decode, NULL,
    ipfib_testsbt_xdr_stat_encode, NULL, ipfib_testsbt_xdr_stat_size,
    ipfib_testsbt_xdr_event_encode, NULL,
    NULL,
    ipfib_testsbt_show_sb
};
```

23.4.2 Managing FIB Subblocks

With these function tables defined for our feature, we are now ready to talk about subblock management.

23.4.2.1 FIB Subblock Data Structures

What is actually stored in a FIB subblock? What is its type? Apart from some FIB control fields at the beginning of the structure, the FIB subblock data structure can be anything, for example:

```

/*
 * IPFIB test subblock
 */
typedef struct ipfib_testsbt_ {
    FIBSWSB_BASE;                      /* Common subblock header */
    /* Test subblock data after here... */
    ulong control;                     /* Something interesting to send */
    ulong counter;                     /* Some stats to sync up */
    ulong si_counter;
    ulong event_type;                 /* Events generated on LC */
} ipfib_testsbt;
```

Note The `FIBSWSB_BASE` is very, very important! It must be present, and it must be the first entry. It expands to a field called `fibswsb_base` of type `fibswsb_t`. This is a structure that contains a set of FIB subblock control fields.

This field and its type is also the method of passing “anonymous” FIB subblocks around the system (which is why it must come first). To typecast from the feature structure to an anonymous FIB subblock structure, you use `&testsbt->fibswsb_base`, and to get from an anonymous FIB subblock back to the feature structure, you use

`(ipfib_testsbt *) fibswsb;`

23.4.2.1.1 Creating, Initializing, and Deleting the FIB Subblock Facility

There are three methods for creating, initializing, and deleting the FIB subblock data structures:

- 1 Configuration
- 2 Asynchronous
- 3 Creation/deletion

The first two are specific to the RP environment. If the feature can create/delete subblocks from configuration changes, then you can use the configuration method. If the feature can create/delete subblocks as the result of some other system event, for example network session creation, then you should use the asynchronous method. On the LC, the third method, creation/deletion, is used: the XDR method. All three methods are now described.

The configuration method of creation/deletion of FIB subblocks on the RP relies on the configuration commands calling either `reg_invoke_fib_update_fibidb()` or `reg_invoke_fib_update_fibhwidb()` once the new configuration has been read. An example of this is the **ip verify unicast reverse-path** command. The function `ip_unicast_rpf_command()` calls `reg_invoke_fib_update_fibidb()`, which causes the configuration method of FIB subblock creation to run. It creates/initializes and deletes FIB subblocks with the helps of four registry functions that need to be defined per FIB subblock type:

- `fibswsb_required`
- `fibswsb_create`
- `fibswsb_update`
- `fibswsb_delete`

```

DEFINE fibswsb_required
/*
 * Per-FIB sw subblock
 * Return true if this subblock is required given the current
 * configuration of the fibidb
 */
RETVAL
boolean
fibidbtype *fibidb
FIBSWSB_MAX
fibswsb_id_t id
END

DEFINE fibswsb_create
/*
 * Per-FIB sw subblock
 * Create and return the subblock data structure
 */
RETVAL
fibswsb_t *
fibidbtype *fibidb
FIBSWSB_MAX
fibswsb_id_t id
END

DEFINE fibswsb_update
/*
 * Per-FIB sw subblock
 * Update the subblock data structure for the current state of the fibidb
 */
CASE

```

```
void
fibswsb_t *swsb, fibidbtype *fibidb
FIBSWB_MAX
fibswsb_id_t id
END

DEFINE fibswsb_delete
/*
 * Per-FIB sw subblock
 * Delete the subblock data structure
 */
CASE
void
fibswsb_t *swsb
FIBSWB_MAX
fibswsb_id_t id
END
```

The `fibswsb_create` and `fibswsb_delete` registry functions are needed on both the RP and LC. The `fibswsb_required` and `fibswsb_update` functions are RP-only.

On the RP, the `fibswsb_required` registry function is called to see if the current state of the `fibidbtype` structure and the current state of the system in general requires a FIB subblock. If the function returns TRUE, indicating that the subblock is required, and if the subblock does not already exist, the `fibswsb_create` registry function is called to create the FIB subblock data structure (normally using `malloc()`) and then link it to the `fibidbtype` structure using `fibswsb_add()`. One of the parameters to `fibswsb_add()` is the FIB subblock function table. (See? I told you we needed to talk about them first.) For the quick thinking among you, this also hints that the `fibswsb_create` registry functions will be different on the RP and LC since the function table will be different (as described in section 23.4.1.8, “Example: FIB Function Tables”).

Once the FIB subblock has been created, the `fibswsb_update` registry function is called. This is to allow the data structure to be set up according to the current state of the system.

If the `fibswsb_required` registry function returns FALSE, indicating that the subblock is not required, then, if the subblock is present, the `fibswsb_delete` registry function is called to unlink the subblock from the `fibidbtype` structure, using `fibswb_delete()`, and then the memory associated with the structure is returned (normally with `free()`).

In a distributed environment, the control data in the FIB subblock is then automatically sent to all line cards that need to have their local copies of the FIB subblock updated.

For the asynchronous method of FIB subblock creation/deletion on the RP, the feature implementor has to do the decision-making manually.

As implementor, you are responsible for creating and deleting the FIB subblocks, using the same sort of code as was used in the `fibswsb_create` and `fibswsb_delete` registry functions above (or even using the registry function directly).

For this method to work in distributed environments, the feature must also make a call to `fibswsb_distribute_control_data()` to cause any creations/deletions or updates to be distributed to all the necessary line cards.

The `fibswsb_required` registry function must again be used to indicate whether the subblock is needed. This is to make deletion of FIB subblocks work on the LCs. To make deletions work, the call to `fibswsb_distribute_control_data()` must be made *before* the FIB subblock is deleted, and the `fibswsb_required()` function must return FALSE.

It is safe to call the `fibswsb_distribute_control_data()` function in a nondistributed environment. The FIB subblock facility will take care of it.

In a distributed environment the `fibsbsb_distribute_control_data()` call, invoked automatically by the configuration method or manually by the asynchronous method, causes XDR messages to be sent to all line cards that need to create/delete/update the mirrored FIB subblock structure. On the LC, this can cause the XDR method of creation/deletion to be invoked.

If a control XDR message arrives with update information, then the mirrored FIB subblock is accessed. If it does not exist, the `fibsbsb_create` registry function on the LC is called. Then the update can take place. If a control XDR message arrives with delete information, then if the mirrored FIB subblock exists, the `fibsbsb_delete` registry function is called to unlink and free the subblock.

Note The `fibsbsb_delete` registry function can be the same on both the RP and LC.

23.4.3 Using FIB Subblocks

Well, now that we've got synchronized FIB subblocks up and running on the RP and LC, how do you actually use them? This is no different from the normal subblock method. When you need to access the data, two methods can be used, the *get* or the *use* method.

The get method returns a pointer to the FIB subblock data structure, for example:

```
/* Access subblock */
testsb = fibsbsb_get(fibidb, FIBSWSB_TEST);
```

Then you simply access the data! If you are running a feature where the FIB subblocks can be created/deleted in a different process, no guarantees are made about whether the pointer remains valid, which brings us to the second, use, method:

```
/* Access subblock */
testsb = fibsbsb_use(fibidb, FIBSWSB_TEST);

/* Play with subblock */
...

/* Release subblock */
fibsbsb_release(fibidb, FIBSWSB_TEST);
```

In this example, usage counters are maintained on the subblock, preventing the deletion of the subblock while the usage is greater than 0 (`fibsbsb_delete()` returns `FALSE`).

The only final comment is, don't play with the `FIBSWSB_BASE` fields.

23.4.3.1 XDR Support Routines for the Distributed Environment

If you are implementing a nondistributed-only feature (shame on you), then you can skip the next few sections, which describe how a feature hooks into the automatic synchronization supported by FIB subblocks.

Synchronization is accomplished by sending data in XDR messages over the IOS IPC mechanism between the RP and LCs. The good news is that you don't need to know anything about XDRs or IPC; the FIB subblock facility takes care of all that. The bad (but not unexpected) news is that the feature must help the FIB subblock facility encode and decode the feature's data in the XDR buffers.

There are three paths of synchronization:

- control

- statistics
- events

The first is RP-to-LC and the rest are LC-to-RP.

23.4.3.1.1 Control Data Synchronization

Control data synchronization allows control data in RP subblocks to be sent to all the mirroring subblocks on the LCs that need to know about any changes. It also controls the creation/deletion of the mirrored subblocks (see discussion above).

On the RP, when a subblock (or subblocks) need to be synchronized, the FIB subblock facility allocates an empty message buffer. For each subblock, calls are made to two functions registered in the subblock's function table: `fibsbsb_xdr_control_size` and `fibsbsb_xdr_control_encode`. The `fibsbsb_xdr_control_size` routine returns the maximum amount of data (in bytes) that the subblock will need to encode in the message buffer. This allows for dynamic sizing of the data. That amount of space is then allocated in the message buffer, and the `fibsbsb_xdr_control_encode` function is called. This routine encodes the data in the buffer at the location specified by a pointer passed to the routine. The routine then updates that pointer to point to the next free location in the buffer. The routine returns TRUE if there was data to send, otherwise it returns FALSE. A typical encode routine might look like this:

```
boolean ipfib_testsbs_xdr_control_encode (fibsbsb_t *sb,
                                         uchar **xdr_buffer_ptr)
{
    uchar *xdr_buffer = *xdr_buffer_ptr;
    ipfib_testsbs *testsbs = (ipfib_testsbs *) sb;

    if (testsbs->control == 0)
        return (FALSE);

    PUTLONG_ADVANCE(xdr_buffer, testsbs->control);
    *xdr_buffer_ptr = xdr_buffer;

    return (TRUE);
}
```

Depending on the setting of the `FIBSB_XXXX_IRQ_SAFE/_UNSAFE` flag in the function table, this routine is either called from process level with interrupts disabled (flag is `_UNSAFE`, indicating the encode routine is not interrupt-safe), or interrupts enabled (flag is `_SAFE`, indicating the encode routine is interrupt-safe).

Three macros are available to help encode short, long, and longlong data values (bytes don't need a macro!). They are: `PUTSHORT_ADVANCE`, `PUTLONG_ADVANCE`, and `PUTLONGLONG_ADVANCE`.

Why not just bcopy the data structure into the buffer? Why not just write the long directly? Using macros means that the RP and LC can be implemented using different system architectures, for example different endianness or different word sizes.

This process is repeated for all subblocks that might need to send control information. The message is then packaged in an XDR and sent to the correct LCs. The LCs are chosen depending on the `fibsbsb_dist_type` specified in the function table.

On the LC, the XDR message is received and the individual FIB subblock data portions are identified. At this point the XDR method of creation/deletion described above takes place. If the message is an update, then the message data is passed to the `fibsbsb_xdr_control_decode` routine. This routine does the reverse of the encode function on the RP, and extracts the new data

and writes it into the FIB subblock data structure. The `xdr_len` parameter is the length of data pointed to by `xdr_buffer_ptr`. This can be used to sanity check the data and to decode multiple messages packed into the same buffer by the encode routine. Here is an example decode routine:

```
void ipfib_testsbt_xdr_control_decode (fibswsb_t *sb, uchar **xdr_buffer_ptr,
                                         ushort xdr_len)
{
    uchar *xdr_buffer = *xdr_buffer_ptr;
    ipfib_testsbt *testsbt = (ipfib_testsbt *) sb;

    /* Read data from control */
    GETLONG_ADVANCE(xdr_buffer, testsbt->control);

    /* Reset event handler */
    testsbt->event_type = 0;

    /* Update XDR buffer location */
    *xdr_buffer_ptr = xdr_buffer;
}
```

Like the encode routine, three macros are provided to read short, long, and longlong data values: `GETSHORT_ADVANCE`, `GETLONG_ADVANCE`, and `GETLONGLONG_ADVANCE`.

Depending on the setting of `FIBSB_XXXX_IRQ_SAFE/_UNSAFE` flag in the function table, this routine is either called from process level with interrupts disabled (flag is `_UNSAFE`, indicating the decode routine is not interrupt-safe), or interrupts enabled (flag is `_SAFE`, indicating the decode routine is interrupt-safe).

23.4.3.1.2 Statistic Data Synchronization

On the line card, every ten seconds, all the FIB subblocks are polled to see if they have any new statistics to be sent to the RP for accumulation. Like the control synchronization, this uses three feature routines to implement, two on the LC and one on the RP.

On the LC, the statistics process allocates a message buffer and then for each subblock calls the `fibswsb_xdr_stat_size` function. This returns the maximum amount of message space (in bytes) that the subblock's statistics would take. The process then calls the `fibswsb_xdr_stat_encode` routine to allow the subblock to write any new statistics into the message buffer. If there are no new statistics, the routine returns FALSE. If there are statistics, the routine writes them in the buffer, updates the last byte written pointer, and returns TRUE. Here is an example encode routine:

```
void ipfib_testsbt_xdr_stat_decode (fibswsb_t *sb, uchar **xdr_buffer_ptr,
                                         ushort xdr_len)
{
    uchar *xdr_buffer = *xdr_buffer_ptr;
    ipfib_testsbt *testsbt = (ipfib_testsbt *) sb;
    ulong count, counter;

    count = testsbt->counter;
    counter = count - testsbt->si_counter;

    if (counter == 0)
        return (FALSE);
    else {
        /* Record and clear counter */
        PUTLONG_ADVANCE(xdr_buffer, counter);
        testsbt->si_counter = count;
```

```

        /* Update XDR buffer location and indicate that data was written */
        *xdr_buffer_ptr = xdr_buffer;
        return (TRUE);
    }
}

```

Depending on the setting of `FIBSB_xxxx_IRQ_SAFE/_UNSAFE` flag in the function table, this routine is either called from process level with interrupts disabled (flag is `_UNSAFE`, indicating the encode routine is not interrupt-safe) or interrupts enabled (flag is `_SAFE`, indicating the encode routine is interrupt-safe).

When all subblocks have written their statistics, the message is put in an XDR buffer and sent to the RP.

On the RP, each block of statistics is passed to the equivalent FIB subblock's decode routine, which can then aggregate the statistics as shown here:

```

void
ipfib_testsbt_xdr_stat_decode (fibswsb_t *sb, uchar **xdr_buffer_ptr)
{
    uchar *xdr_buffer = *xdr_buffer_ptr;
    ipfib_testsbt *testsbt = (ipfib_testsbt *) sb;
    ulong counter;

    /* Counters */
    GETLONG_ADVANCE(xdr_buffer, counter);
    testsbt->counter += counter;

    /* Update XDR buffer location */
    *xdr_buffer_ptr = xdr_buffer;
}

```

Depending on the setting of `FIBSB_xxxx_IRQ_SAFE/_UNSAFE` flag in the function table, this routine is either called from process level with interrupts disabled (flag is `_UNSAFE`, indicating the decode routine is not interrupt-safe), or interrupts enabled (flag is `_SAFE`, indicating the decode routine is interrupt-safe).

23.4.3.1.3 Event Data Synchronization

The final synchronization facility that exists allows the LC to generate asynchronous events on subblocks that get reported back to the equivalent subblock on the RP. A call `fibswsb_send_event_data()` for a FIB subblock on the LC initiates the process.

The actual encoding/decoding of the event data is made slightly simpler than the statistics synchronization, because only a single event is passed in each XDR message. So the FIB subblock facility does not need to find the size.

On the LC, when an event is generated using the `fibswsb_send_event_data()` routine, the FIB subblock facility allocates a message buffer and calls the `fibswsb_xdr_event_encode` routine for the subblock that raised the event. The routine is identical in function to the statistics encode routine. Here is an example:

```

void
ipfib_testsbt_xdr_event_decode (fibswsb_t *sb, uchar **xdr_buffer_ptr,
                                ushort xdr_len)
{
    uchar *xdr_buffer = *xdr_buffer_ptr;
    ipfib_testsbt *testsbt = (ipfib_testsbt *) sb;

```

```

/* Write event */
PUTLONG_ADVANCE(xdr_buffer, testsb->event_type);

/* Update XDR buffer location, and indicate that data was written */
*xdr_buffer_ptr = xdr_buffer;
return (TRUE);
}

```

Note, the routine could return FALSE, which would cause the event to be cancelled.

Depending on the setting of `FIBSB_xxxx_IRQ_SAFE/_UNSAFE` flag in the function table, this routine is either called from process level with interrupts disabled (flag is `_UNSAFE`, indicating the encode routine is not interrupt-safe), or interrupts enabled (flag is `_SAFE`, indicating the encode routine is interrupt-safe).

When the XDR buffer is properly encoded, the FIB subblock facility sends it to the RP.

On the RP, the subblock to receive the event is found, and the event message is passed to the decode routine. Here is an example:

```

void
ipfib_testsbs_xdr_event_decode (fibswsb_t *sb, uchar **xdr_buffer_ptr)
{
    uchar *xdr_buffer = *xdr_buffer_ptr;
    ipfib_testsbs *testsbs = (ipfib_testsbs *) sb;

    /* Get event from LC */
    GETLONG_ADVANCE(xdr_buffer, testsbs->event_type);

    /* Deal with event! */
    buginf("\nKewl, received event (type=%d) for %s",
           testsbs->event_type, sb->fibswsb_idb->namestring);

    /* Update XDR buffer location */
    *xdr_buffer_ptr = xdr_buffer;
}

```

Depending on the setting of `FIBSB_xxxx_IRQ_SAFE/_UNSAFE` flag in the function table, this routine is either called from process level with interrupts disabled (flag is `_UNSAFE`, indicating the decode routine is not interrupt-safe), or interrupts enabled (flag is `_SAFE`, indicating the decode routine is interrupt-safe).

23.4.4 Debugging Subblocks

OK, you've implemented your feature using subblocks, but it doesn't seem to work. So now what? First, there is a command to show the current state of the subblocks attached to `fibidbtype` and `fibhwidb` type structures:

show cef interface [interface] internal

Be warned, the **internal** keyword is hidden, since we don't want every Tom, Dick, and Harry seeing the internal data contents of our structures. The optional *interface* parameter specifies which interface should be displayed. Without the parameter, all interfaces are shown.

The *interface* parameter actually refers to a `fibidbtype` structure, so to see the `fibhwidbtype` subblocks, the command finds the parent `fibhwidbtype` and, for every subblock present, calls the `fibhwidb_show_sb_t` function installed in the function table for that subblock ID. If no routine is present, then the subblock ID name is printed along with `No decode` to show that the subblock is at

least present. The command then walks the list of subblocks attached to the specified `fibidbtype` and calls the `fibsbsb_show_sb_t` function for each one. The same output is shown if the no decode routine is present.

Example:

```
ed6-75a#sho cef interface eth0/1/1 internal
  Ethernet0/1/1 is up (if_number 4)
    Internet address is 192.168.200.1/24
    ICMP redirects are always sent
    Per packet loadbalancing is disabled
    IP unicast RPF check is enabled
    Inbound access list is not set
    Outbound access list is not set
    IP policy routing is disabled
    Hardware idb is Ethernet0/1/1
    Fast switching type 1, interface type 5
    IP Distributed CEF switching enabled
    IP Feature Fast switching turbo vector
    IP Feature CEF switching turbo vector
    Input fast flags 0x4000, Output fast flags 0x0
    ifindex 3(3)
    Slot 0 Slot unit 9 VC -1
    Transmit limit accumulator 0x48001A12 (0x48001A12)
    IP MTU 1500
    Subblocks:
      unicast RPF: acl=0, drop=0, sdrop=0
      test2: no decode
      test: control=0xC0A8C801, counter=35 (35/0)
```

This command works on both the RP and LC.

The second line of attack is the FIB subblock **debug** command:

```
debug ip cef subblock [ id (all|sw|hw) id ] [ xdr (all|control|statistics|event) ]
```

In its simplest form, all FIB subblock events (creations / deletions / XDR messages / errors) for all subblock types will be printed. Again, this command can be run on both the RP and LC. As more and more subblocks are added, this output will start to become too overwhelming, so the command allows you to pick what subblock ID to include, and what XDR messages.

Example:

```
debug ip cef subblock - debug everything
debug ip cef id all xdr control - show only XDR control messages
debug ip cef id sw 1 - debug only SW subblock id 1
```

The ID numbers come from the `fibsbsb_id_t` enum you defined right at the beginning of this process. Go and count (from 0) to find your feature's ID number.

23.4.4.1 Frequently Asked Question

Q. Why is the control encode routine being called twice?

A. This is a result of supporting both methods of FIB subblock create/update/delete; configuration change and asynchronous change. Here are two tracebacks to explain:

First traceback:

```
0x60067368:l2tp_fibswsb_xdr_control_encode(0x6006733c)+0x2c
0x60B94410:do_fibidb_dist_sb_control(0x60b9412c)+0x2e4
0x60B94648:fibidb_dist_sb_control(0x60b945fc)+0x4c
0x6007ACC0:fibswsb_distribute_control_data(0x6007ac9c)+0x24
0x600677FC:l2tp_ipfib_sb_create(0x60067718)+0xe4
0x60631C78:vpdn_session_up(0x60631bec)+0x8c
0x6064B3C0:l2tp_recv_ICRP(0x6064b32c)+0x94
0x60646A70:l2tp_process_control_packet(0x606462b0)+0x7c0
0x60647A8C:l2tp_mgmt_daemon(0x60647804)+0x288
0x6015D6CC:r4k_process_dispatch(0x6015d6b8)+0x14
0x6015D6B8:r4k_process_dispatch(0x6015d6b8)+0x0
```

Second traceback:

```
0x60067368:l2tp_fibswsb_xdr_control_encode(0x6006733c)+0x2c
0x60B94410:do_fibidb_dist_sb_control(0x60b9412c)+0x2e4
0x60B94648:fibidb_dist_sb_control(0x60b945fc)+0x4c
0x60B94B90:fibidb_collect_subblock_control(0x60b94b80)+0x10
0x6008BE70:fibidb_swif_comingup(0x6008bd88)+0xe8
0x60142B74:transition_adjust(0x60142a34)+0x140
0x6013A690:handle_transition(0x6013a490)+0x200
0x6013B2D8:net_background(0x6013b10c)+0x1cc
0x6015D6CC:r4k_process_dispatch(0x6015d6b8)+0x14
0x6015D6B8:r4k_process_dispatch(0x6015d6b8)+0x0
```

The first traceback shows a FIB subblock being asynchronously created and added to a fibidb. The control data is encoded and sent to the line cards.

The second traceback shows the same fibidb coming up at a later point. This causes the FIB subblock to be updated due to the configuration change, and causes the second control message to be encoded and sent to the line cards. The control data may or may not be identical in both cases, depending on the feature being implemented.

So even if the FIB subblocks are intended only to be controlled via the asynchronous method, any configuration changes (including if up/down) will cause additional control messages to be passed to the line cards. The decode routine needs to be coded to be aware of this possibility.

23.4.4.2 Further Help and Support

FIB subblocks were implemented by Neil Jarvis, njarvis@cisco.com, and are supported by the FIB team: interest-fib@cisco.com. Bug fixes, additional code, enhancements, or suggestions are always welcome.

23.5 Hardware Session and L2 Hardware Switching

The platform interface defined by the Hardware Session API and the L2 Hardware Switching API (*Both new in 12.3T*) provides the baseline infrastructure and feature set needed to support the next generation AAA, SSS, and SSG. The information in this section is based on EDCS-163630 Version 1.1, “HW Switching Abstraction API for Tunneling Protocols”.

Note Information on the latest infrastructure and APIs that are planned to be committed in 12.2S RLS7 (and ultimately in 12.3T) is in EDCS-329024 and EDCS-329026.

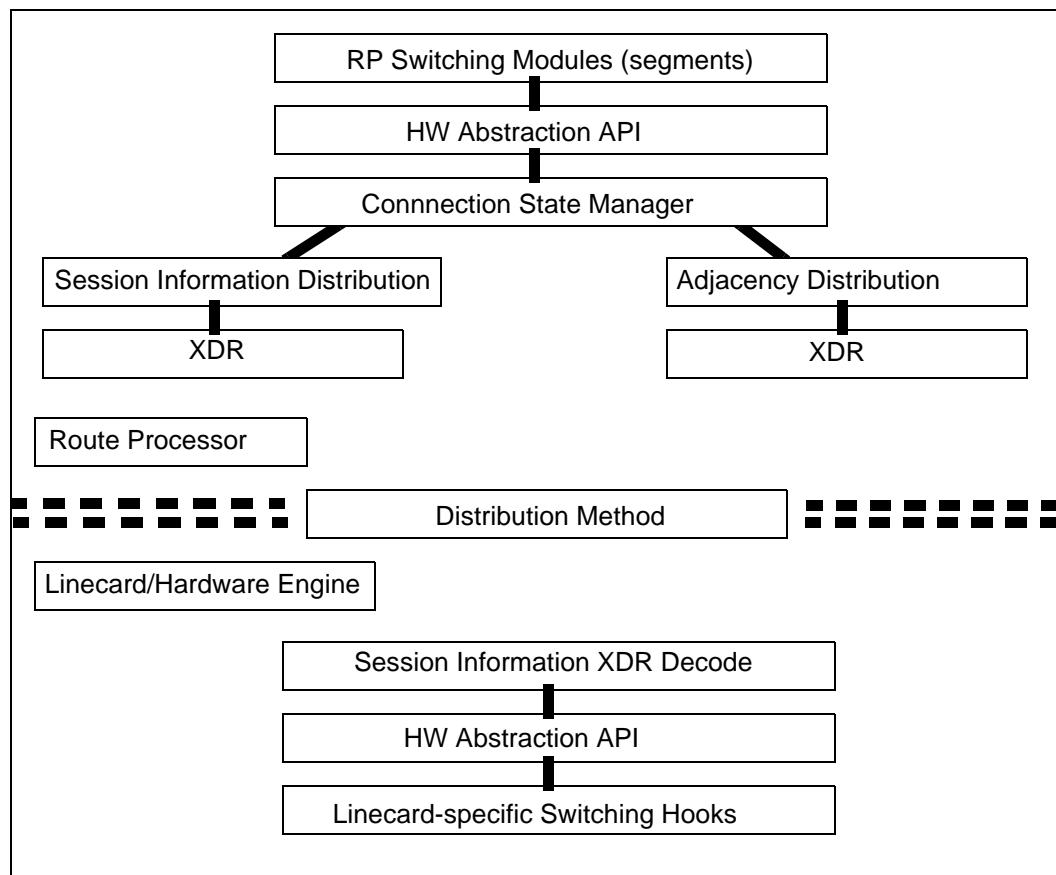
23.5.1 Background

More and more platforms are adding hardware support for special features such as L2TP (Layer 2 Tunneling Protocol), PPPoE, PPPoA, and other tunneling protocols. Just from a L2TP perspective, there have been at least three implementations of hardware-based switching for L2TP packets. The c10K (toaster), Marvel (dcef), and NSE-1 (toaster) all support some form of distributed switching using either hardware or software. Each of these platforms handles the switching setup differently. In many cases, code was added in the form of registries from the VPDN code to the switching drivers. In the case of Marvel, dCEF subblocking was used to distribute L2TP information down to the linecards. The iWAN project requires L2TP hardware switching to be supported in at least two more distributed platforms, the 7500 and the GSR. There are also the designs that need to be supported where IDBs are not present to serve as the common communication handle, as in the case of multipoint VCs on top of ATM. All of these platforms have their nuances regarding how the switching is handled and how the state is distributed from the RP to the linecard or hardware engines. The Hardware Session and L2 Hardware Switching registry functions were developed to provide the HW Abstraction API, which is an interface between the software modules and the hardware drivers so that L2 hardware switching is manageable both now and in the future.

23.5.1.1 Architectural Models Supported by the HW Abstraction API

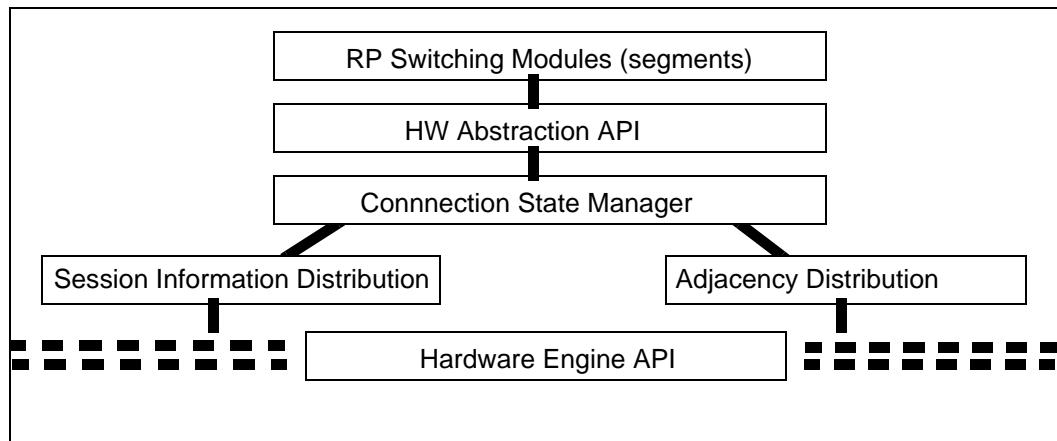
In general, the model for this architecture when there are linecards is shown in Figure 23-26:

Figure 23-26 L2 Hardware Switching Support for an Architecture with Linecards



In general, the model for this architecture when there is a hardware engine (that is toaster) is shown in Figure 23-27:

Figure 23-27 L2 Hardware Switching Support for an Architecture with a Hardware Engine



The platform-independent modules above the HW Abstraction API are required to advertise events such as session up and down, and provide change notifications on established sessions. The platform-independent modules also are expected to be able to obtain counters via the HW Abstraction API.

The platform-dependent code that sits below the HW Abstraction API is expected to process the session information and distribute the necessary information to the linecards or hardware engines. The platform-dependent code will need to maintain counters for the sessions and must be able to provide these counters upon request. The platform-dependent code must also be able to handle repopulation. Repopulation deals with the RP redistributing state to each of the linecards or hardware engines. This would typically be due to an event such as OIR, linecard reset, or dCEF enable. When an event such as this occurs, the RP must redistribute all of the current state from the RP down to each of the linecards or hardware engines.

The HW Abstraction API has the following requirements and definitions:

- It must be implemented as libraries; assume run-to-completion APIs and assume that no processes are needed. IPC/processes can be used for the platform-specific distribution, but this all happens underneath the L2 Hardware Switching API.
- No shared memory can exist between the platform-independent software switching and hardware switching subsystems. Opaque handles should be used to represent the state communicated through the API.

23.5.1.1.1 Common Definitions for Both Local Termination and L2-L2 Switching

The maximum number of bytes for the rewrite macstring is set to 48:

```
#define HW_MAX_MACSTRING_LEN 48
```

23.5.1.1.2 Common Counter Block

The HW Abstraction API assumes that there will exist some mechanism on either the line card or the hardware engine for obtaining counters per segment or session. Regardless of the method, the counters are reported via the following structure:

```
typedef struct {
    ulong rcvd_bytes;
    ulong rcvd_pkts;
    ulong rcvd_drops;
    ulong xmit_bytes;
    ulong xmit_pkts;
    ulong xmit_drops;
} hw_switch_counters;
```

23.5.1.1.3 The hw_switching_down() Callback Mechanism

This API is used as a callback mechanism to inform the clients of the HW Abstraction API that a segment or session is now down. This can occur as the result of an OIR event, linecard reset or failure.

If the `hw_switch_rc` is `HW_SWITCH_DOWN_DROP`, the client should perform the appropriate action to tear down the session associated with the context:

```
typedef void (*hw_switching_down)(void *context, hw_switch_rc);
```

The following subsections provide information on the APIs that were added to the IOS infrastructure to provide the support needed for the two key scenarios which need to be addressed when considering a hardware switching design:

- “23.5.2 Support for Locally-Terminated Tunnels”
- “23.5.3 Support for Layer 2 to Layer 2 Point-to-Point Connections”

23.5.2 Support for Locally-Terminated Tunnels

Locally-terminated tunnels are typically associated with a virtual interface. Virtual interfaces are either “int tunnel” or “virtual-access” interfaces. In this scenario, packets arrive via an interface and are handed to the tunnel decapsulation code. After identifying the corresponding session state, the packet is prepared for a layer 3 forwarding decision. The packet is given to the IP forwarding code to determine what features need to be applied to the packet and what egress interface the packet should be sent out on.

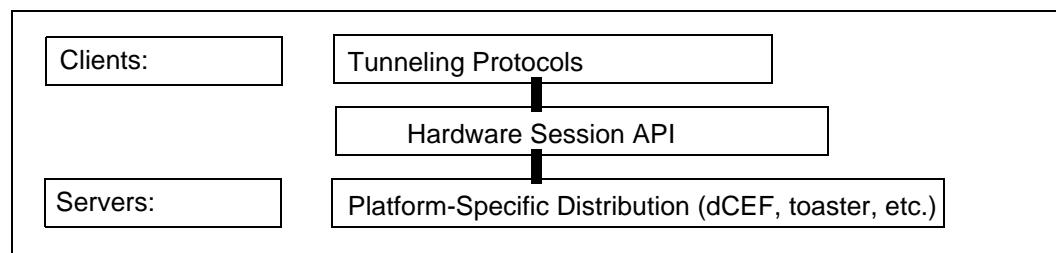
Likewise, forwarding entries in the IP FIB table determine when packets should be transmitted out the virtual interface. Even today, forwarding and adjacency information is distributed using hooks in the CEF adjacency/FIB code. This mechanism is well-known and is supported in some fashion on every platform. The only work that really needs to occur here is to have an API that allows the signalling protocol code, that is L2TP, PPPoE, PPPoA, etc., to advertise new sessions and the corresponding features for these sessions to the hardware layer. This session information needs to be associated with the adjacency information on the hardware engines so that the correct packet fixup and features can be applied to transmitted packets. For received packets, the session information is needed so that the correct features and virtual-interface information can be applied to the incoming packets.

23.5.2.1 The Hardware Session API

Support for locally-terminated tunnels is provided by the Hardware Session API. The Hardware Session API is used to advertise information about session creation and updates. It is also used to obtain counters from the hardware. In general, the sessions advertise that information is needed to bind session information (that is associated with a tunneling protocol) to a dynamically-created virtual interface. This advertisement is needed so that the hardware can receive and process packets from the tunneling protocol and associate those packets with the correct interface. Adjacency and forwarding information takes care of setting up the transmit path for these virtual interfaces; however, session information is needed to provide transmit features and adjacency fixup settings.

The general architecture for this API is shown in Figure 23-28:

Figure 23-28 Hardware Session API Architecture



The `hw_session_handle` is an opaque handle allocated by the platform-specific hardware distribution engine. It is used for maintaining state regarding the session within the distribution engine. Any platform distribution engine that plugs underneath this API must be able to allocate state and associate that state with this handle. Any client of this API must be able to store this handle in their relevant context.

```
typedef struct hw_session_handle_ hw_session_handle;
```

This enumeration defines the various types of protocols which can advertise their session information to the hardware:

```
typedef enum {
    HW_SESSION_L2TP = 1,
    HW_SESSION_L2F,
    HW_SESSION_PPPOE,
    HW_SESSION_PPPOA,
} hw_session_type;
```

The `hw_session_info` represents the information about the tunnel/session. The `session_info` element is a variable length byte array that should be used to identify session specific attributes which must be distributed to the hardware switching engines. The platform-specific distribution engine is responsible for decoding this information by type-casting the session information to the appropriate data structure based on the `hw_session_type`.

```
typedef struct {
    hw_session_type session_type;
```

The `macstring` represents the information that should be prepended to the packet on transmit. The `session_info` will contain the feature and fixup information for how the packet should be handled during transmit and receive processing.

Note If the hardware is only handling IP packets, the macstring can be ignored because the adjacencies that are pushed from CEF will contain the macstring.

```
uchar macstring_len; /* Length of the macstring in bytes */
uchar macstring[HW_MAX_MACSTRING_LEN];

ushort session_info_len; /* Length of the session_info in bytes */
uchar session_info[0];
} hw_session_info;
```

The `hw_session_change_info` represents information about the tunnel/session that has changed since the last session open/update event. The platform-specific distribution engine is responsible for decoding this information by type-casting the session information to the appropriate data structure based on the `hw_session_type`. The platform-specific distribution code must determine how to propagate this information down to the hardware engine or linecard(s).

```
typedef struct {
    hw_session_type session_type;

    ushort session_info_len; /* Length of the session_change_info in bytes */
    uchar session_change_info[0];
} hw_session_change_info;
```

23.5.2.1.1 List of Functions that Support the Hardware Session API

The following registry functions were added to support the Hardware Session API (see the reference pages for detailed information on these functions):

- 1 `reg_invoke_hw_session_clear_counters()`
- 2 `reg_invoke_hw_session_get_counters()`
- 3 `reg_invoke_hw_session_open()`
- 4 `reg_invoke_hw_session_remove()`
- 5 `reg_invoke_hw_session_update()`

23.5.3 Support for Layer 2 to Layer 2 Point-to-Point Connections

In this scenario, packets arrive on some type of attachment circuit and are forwarded out onto some type of tunneling protocol. Likewise, packets received from the tunnel are sent directly out the associated attachment circuit. In this scenario, there are no adjacencies or IP forwarding decisions. In the simplest terms, there is a point-to-point connection between the attachment circuit and the pseudowire. Before the L2 Hardware Switching API, there wasn't a well-defined mechanism for distributing this type of information to hardware engines or linecards. Depending on the platform, dCEF or direct hardware manipulation has been used. The L2 Hardware Switching API was defined to be a generic API that allows the information that is needed to switch packets from the attachment circuit to the pseudowire to also be distributed to the respective linecards or hardware engines.

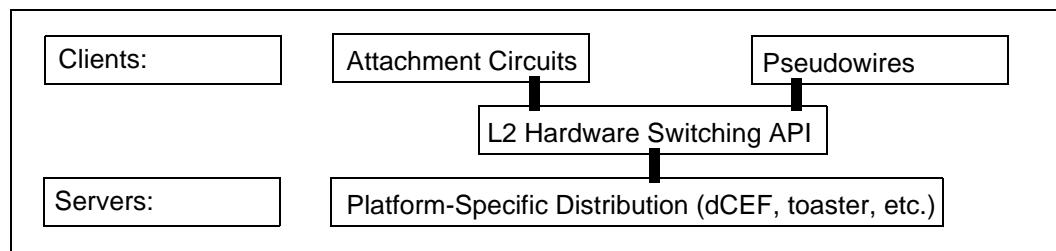
23.5.3.1 The L2 Hardware Switching API

Support for L2 to L2 point-to-point connection switching is provided by the L2 Hardware Switching API. The L2 Hardware Switching API is defined as the centralized API for the software forwarding plane to advertise information regarding Layer 2 switched circuits. The initial use for this API is to

setup the necessary switching information needed to transfer packets from an attachment circuit, i.e. Frame Relay, PPP, Ethernet, ATM, or HDLC, to a pseudowire. The pseudowire is either an IP or MPLS tunneling technology that emulates the properties of the real attachment circuit.

The general architecture for this API is shown in Figure 23-29:

Figure 23-29 L2 Hardware Switching API Architecture



Note There is nothing about this API that prohibits distributed information about local to local switched circuits, that is Frame Relay DLCI to Frame Relay DLCI.

`l2hw_switch_handle` and `l2hw_switch_segment` are opaque handles allocated by the L2 Hardware Switching distribution server for maintaining the state regarding the entire connection and each segment of the connection. Any hardware distribution engine that plugs in underneath this API must be able to allocate the state and associate that state with these handles. Any client of this API must be able to store these handles in their relevant context.

Note All of the L2 Hardware Switching registries are implemented as STUB registries.

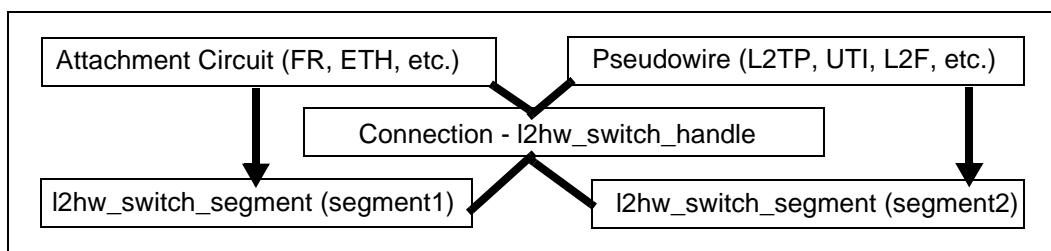
The following structures are for the opaque handles:

```

typedef struct l2hw_switch_handle_ l2hw_switch_handle;
typedef struct l2hw_switch_segment_ l2hw_switch_segment;
typedef struct l2hw_feature_handle_ l2hw_feature_handle;
  
```

The general relationship of these handles is shown in Figure 23-30:

Figure 23-30 Relationship Between the `l2hw_switch_handle` and `l2hw_switch_segment` Handles



This enumeration defines the various types of segments that can be provisioned for and maintained using the L2 Hardware Switching API:

```
typedef enum {
    L2HW_FRAME_RELAY = 1,
    L2HW_ETHERNET,
    L2HW_ATM,
    L2HW_HDLC,
    L2HW_PPP,
    L2HW_PPPOE,
    L2HW_PPPOA,
    L2HW_L2TP,
    L2HW_L2TPv3,
    L2HW_L2F,
    L2HW_UTI,
} l2hw_segment_type;
```

Information about the segment is contained within the following structure. The `segment_info` element is a variable length byte array that should be used by each segment to identify segment-specific attributes that must be distributed to the hardware switching engines. The hardware module that decodes this information is responsible for type-casting the segment information to the appropriate data structure.

```
typedef struct {
    l2hw_segment_type segment_type;

    uchar macstring_len; /* Length of the macstring in bytes */
    uchar macstring[HW_MAX_MACSTRING_LEN];

    ushort segment_info_len; /* Length of the segment_info in bytes */
    uchar segment_info[0];
} l2hw_segment_info;
```

The `l2hw_segment_change_info` structure is used to advertise information about the segment that has changed since the first provision call or the last update call. The `segment_change_info` element is a variable length byte array that should be used by each segment to identify segment-specific attributes that must be distributed to the hardware switching engines. The hardware module that decodes this information is responsible for type-casting the segment change information to the appropriate data structure.

```
typedef struct {
    l2hw_segment_type segment_type;

    ushort segment_info_len; /* Length of segment_change_info in bytes */
    uchar segment_change_info[0];
} l2hw_segment_change_info;
```

Note Clients should only add elements that can actually change to their segment-specific `change_info`.

23.5.3.1.1 L2 Hardware State Machine Description

Here is the generalized state machine description for the platform-specific server that plugs in underneath the L2 Hardware Switching API:

States:

- IDLE: Waiting for new connection
- DOWN: One of the 2 segments is open
- OPEN: Both segments are provisioned
- FREE: Free both segments and connection

Events:

- add: Call to provision a segment
- upd: Call to update an existing provisioned segment
- del: Call to remove a previously provisioned segment
- get: Call to obtain counters from a provisioned segment
- clr: Call to clear counters from a provisioned segment

Actions:

- seg1: Provision segment 1
- seg2: Provision segment 2
- upda: Update segment 1 or 2
- fre1: Free one segment
- fre2: Free other segment and connection
- cntr: Get counters
- ccnt: Clear counters
- inva: Invalid event, print errmsg

Here is the State Machine:

	add	upd	del	get	clr
IDLE:	DOWN	IDLE	IDLE	IDLE	IDLE
DOWN	OPEN	DOWN	FREE	DOWN	DOWN
OPEN	OPEN	OPEN	DOWN	OPEN	OPEN

Here is the Action Table:

	add	upd	del	get	clr
IDLE:	seg1	inva	inva	inva	inva
DOWN	seg2	upda	fre2	cntr	ccnt
OPEN	inva	upda	fre1	cntr	ccnt

L2 Hardware Detailed Action Description

The following pseudo-structure is used to describe the overall actions:

```

typedef struct {
    l2hw_switch_segment segment1;
    l2hw_switch_segment segment2;
} l2hw_connection;

typedef struct {
    l2hw_switch_handle handle;
    l2hw_segment_info segment_info;
    l2hw_counters counters;
} l2hw_segment;

seg1: Provision segment 1
1) Allocate l2hw_connection struct
2) Allocate l2hw_switch_handle, associate handle with
   l2hw_connection object
3) Allocate l2hw_switch_segment handle
4) Allocate l2hw_segment and associate it with the l2hw_switch_segment
   handle.
5) Store segment info into l2hw_segment
6) Associate l2hw_switch_segment with l2hw_connection->segment1
7) Perform platform specific distribution for segment1

seg2: Provision segment 2
1) Locate l2hw_connection associated with the l2hw_switch_handle
2) Allocate l2hw_switch_segment handle
3) Allocate l2hw_segment and associate it with the l2hw_switch_segment
   handle.
4) Store segment info into l2hw_segment
5) Associate l2hw_switch_segment with l2hw_connection->segment2
6) Perform platform specific distribution for segment2
7) Perform any "connection up" processing

upda: Update segment 1 or 2
1) Locate the l2hw_segment associated with the l2hw_switch_segment
   handle.
2) Compare new l2hw_segment_info with the old
   l2hw_segment->segment_info to determine what changed.
3) Update l2hw_segment->segment_info as necessary.
4) Distribute changes as necessary.

fre1: Free one segment
1) Locate the l2hw_segment associated with the l2hw_switch_segment
   handle.
2) Free the segment memory and the segment handle.
3) Distribute the "connection down" information as necessary.

fre2: Free other segment and connection
1) Locate the l2hw_segment associated with the l2hw_switch_segment
   handle.
2) Free the segment memory and the segment handle.
3) Free the l2hw_connection and l2hw_switch_handle.
4) Distribute the "connection removed" information as necessary.

```

```

cntr: Get counters
    1) Locate the l2hw_segment associated with the l2hw_switch_segment
       handle.
    2) Obtain last posted counters for the segment or pull the counters
       directly from the hw engine. Return these counters to the caller.

ccnt: Clear counters
    1) Locate the l2hw_segment associated with the l2hw_switch_segment
       handle.
    2) Clear the counters for the segment.

inva: Invalid event, print errmsg
    1) Print buginf or errmsg

```

23.5.3.1.2 Adding Features to the Hardware Switch

As the hardware switches become capable of doing L2 switching, they must also be able to support features installed on the switching path. The `reg_invoke_l2hw_install_feature()`, `reg_invoke_l2hw_update_feature()`, `reg_invoke_l2hw_remove_feature()`, and `reg_invoke_l2hw_get_feature_report()` functions define a common API to install and manage switching features in a hardware-independent manner. These APIs must be used *after* the hardware switch is provisioned for because they rely on the `l2hw_switch_segment` allocated by `reg_invoke_l2hw_provision_segment()`.

This enumeration defines the various types of features that can be installed using the L2 Hardware Switching API:

```

typedef enum {
    L2HWF_UNKNOWN = 0, /* Invalid type */
    L2HWF_IDLE_PPP,
} l2hw_feature_type;

```

The `l2hw_feature_config_info` structure shown below has two uses. The first is to send feature-specific configuration information to the hardware module. The second is to retrieve feature-specific output from the hardware module. The `feature_info` element is a variable length byte array which should be used to carry feature-specific attributes to/from the hardware module. The feature module and hardware module must know what type of data to expect for a given feature type and type-cast the feature information as needed.

```

typedef struct {
    l2hw_feature_type feature_type;

    ushort feature_info_len; /* Length of feature_info in bytes */
    uchar feature_info[0];
} l2hw_feature_info;

```

23.5.3.1.3 List of Functions that Support the L2 Hardware Switching API

The following registry functions were added to support the L2 Hardware Switching API (see the reference pages for detailed information on these functions):

- 1 `reg_invoke_l2hw_clear_segment_counters()`
- 2 `reg_invoke_l2hw_get_feature_report()`
- 3 `reg_invoke_l2hw_get_segment_counters()`
- 4 `reg_invoke_l2hw_install_feature()`
- 5 `reg_invoke_l2hw_provision_segment()`

```

6 reg_invoke_l2hw_provision_switch()
7 reg_invoke_l2hw_remove_feature()
8 reg_invoke_l2hw_unprovision_segment()
9 reg_invoke_l2hw_unprovision_switch()
10 reg_invoke_l2hw_update_feature()
11 reg_invoke_l2hw_update_segment()

```

23.5.4 Segment and Session Information Handling Details

In both the Hardware Session API and the L2 Hardware Switching API, there exists information in the `hw_session_info` and `l2hw_segment_info` structures that must be interpreted by the platform-dependent code. Specifically, the meat of the information in these structures is contained within the `session_info` and `segment_info` elements of these structures. These are defined as zero-byte arrays to provide an abstract way of identifying per-session information. The zero-byte array is an easy way of defining a contiguous data structure using a modular method. It allows the HW Abstraction API to be stand-alone because the segment-specific data structures and header files do not have to be defined within the API. However, it does force the platform-specific code to follow some specific rules. These rules are:

- The `segment_info` and `session_info` data structures *must* be assumed to be contiguous data structures. This allows them to be easily packaged in an IPC message which can be sent across the backplane to a linecard or sent to an internal process to be handled. To determine the size of this data structure, the following method can be used:

```
message_len = sizeof(l2hw_segment_info) + l2hw_seginfo->segment_info_len;
```

- To interpret the `session_info` or `segment_info` data structures, the server of this API must first lookup at that `session_type` or `segment_type` elements to determine what structure is being represented. Then, the server must perform the appropriate casting of this data, i.e:

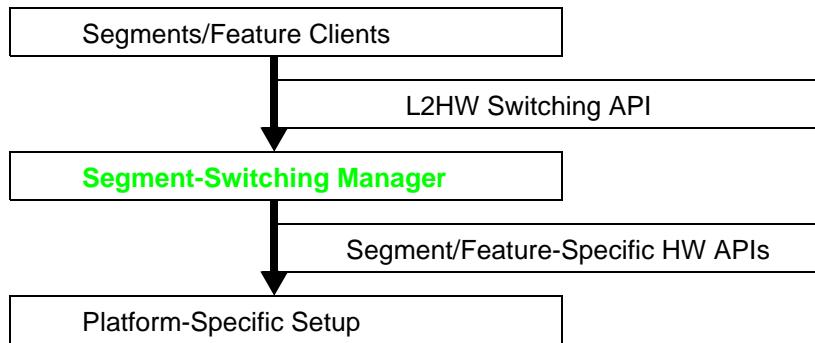
```
if (l2hw_segment_info->segment_type == L2HW_L2TPv3) {
    l2x_info = (l2x_switch_info *)&l2hw_segment_info->segment_info;
}
```

- Since parameters are passed down in a segment or session structure as a whole, the platform-specific code must maintain its own representation of this information to facilitate the session/segment update calls. Whenever the `reg_invoke_l2hw_update_segment()` or `reg_invoke_hw_session_update()` APIs are called, the server must interpret the update with respect to the previously cached information.

23.6 L2VPN Platform API

In 12.2S RLS7, a data-plane process named the Segment Switching Manager (SSM) was plugged into the L2HW (Layer 2 Hardware) Switching API (described in section 23.5, “Hardware Session and L2 Hardware Switching”), between the L2 Hardware Switching API and the hardware, as shown in Figure 23-31.

Note The information in this section is based on EDCS-329026 Rev 4, “LPVN Platform APIs Software Design Specification”.

Figure 23-31 Position of SSM in L2 Switching Architecture

This section defines the mechanism that platforms will use to set up L2VPN services. The following lists show the combinations of services that are handled by the SSM.

Current combinations:

- AC (Attachment Circuit) to L2TP (Layer 2 Tunneling Protocol)
- AC to MPLS (Multi-Protocol Label Switching)
- AC to AC

Future combinations:

- MPLS to MPLS
- L2TP to MPLS
- L2TP to L2TP
- AC to VPLS (Virtual Private LAN Service)
- MPLS to VPLS
- L2TP to VPLS
- MPLS to Virtual Ifc (Virtual Interface, for example, a virtual-access or tunneling interface)
- L2TP to Virtual Ifc

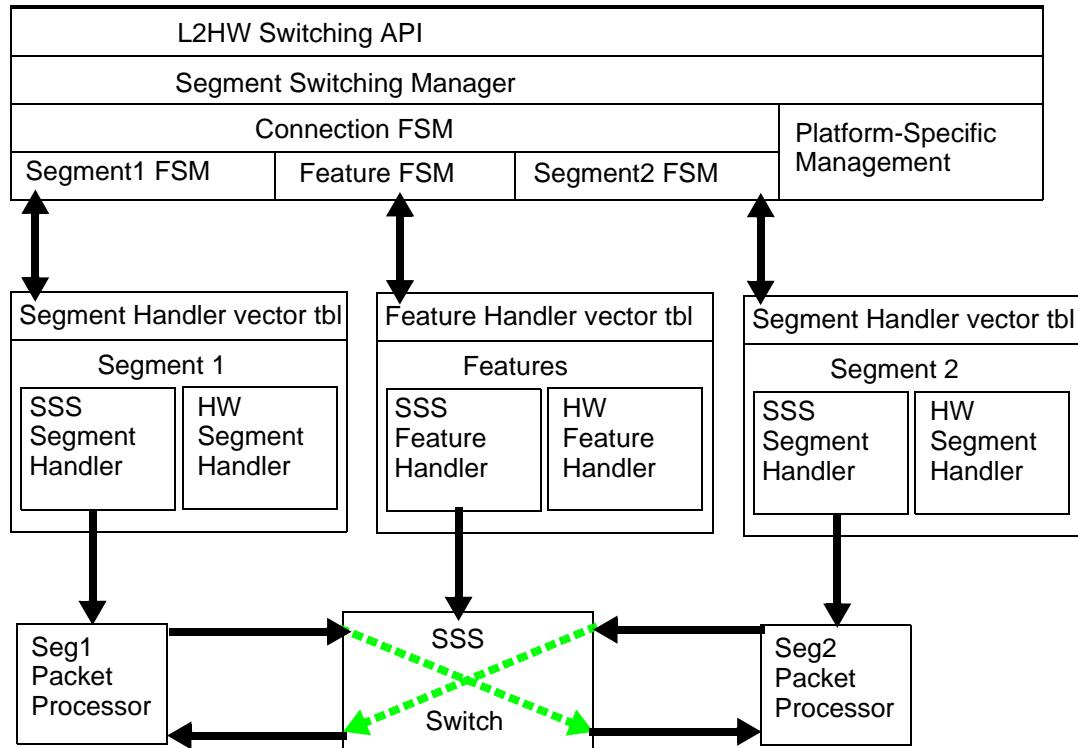
Also, the interworking information needed for the any-to-any feature is advertised through the SSM API. For modularity, any given segment only advertises information specific to its segment.

Therefore, it is not the explicit responsibility of any one segment to advertise information about the other segment. Instead, the APIs provide accessor functions to obtain the handle of the other segment. Using the handle of the other segment, the platform can “pull” the necessary information about the other segment. This provides the platform with the ability to coalesce the information about each segment into one object underneath the platform APIs. This modularity allows the common control-plane to reuse the data-plane setup code in the various combinations listed above. You can consider the APIs described in this section to employ a “2-sided” provisioning model.

23.6.1 Primary Components of SAL

The SSM is the main portion of the overall architecture defined by SAL (Segment-switching Abstraction Layer) and is shown in Figure 23-32.

Figure 23-32 SAL Architecture



The primary components of SAL that are shown in Figure 23-32 are as follows:

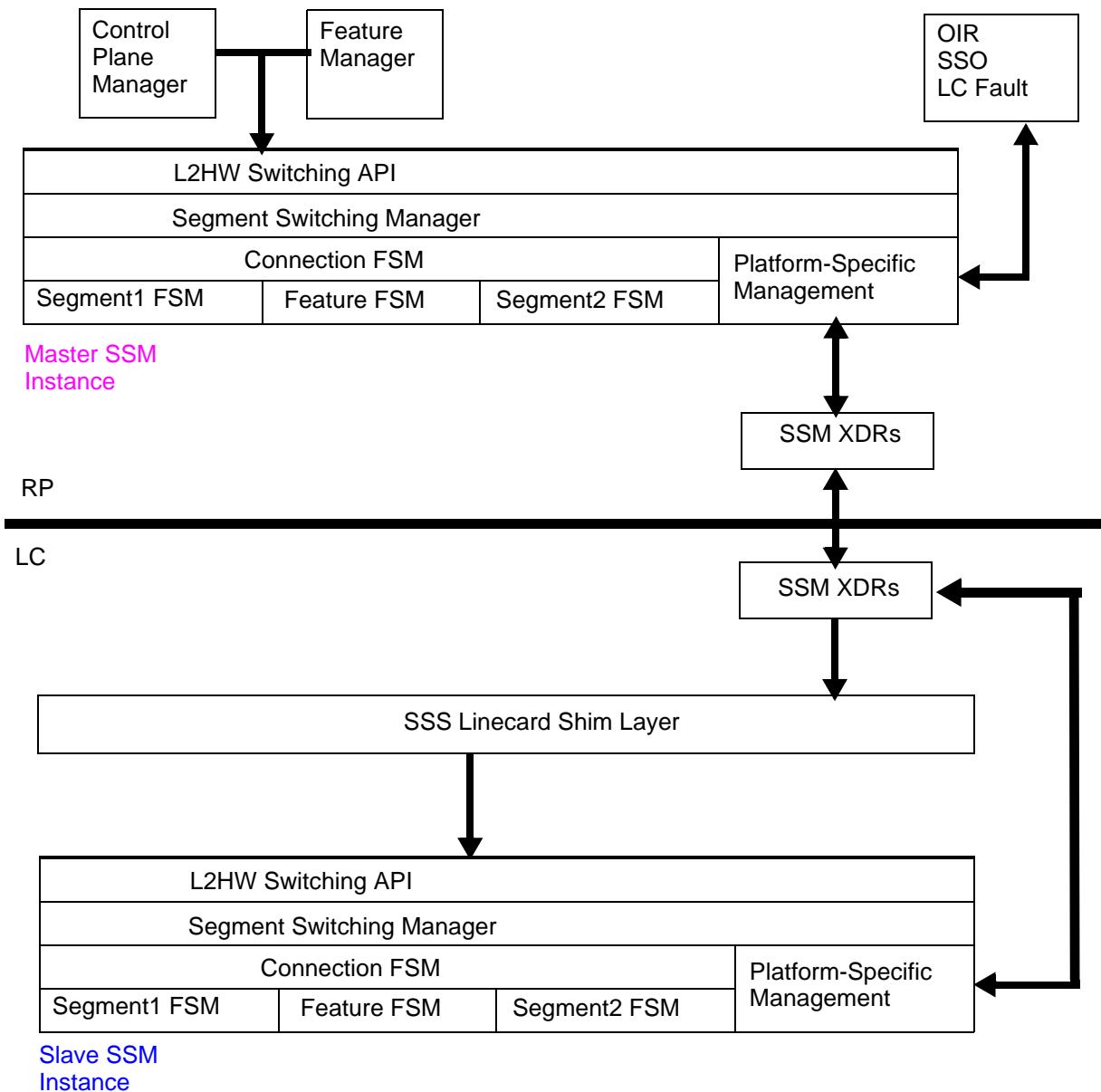
- L2HW Switching API—This represents the APIs between the control-plane processes such as AToM Manager, AC Manager, Feature Manager, and L2TP Manager. The control-plane processes that interface into this API are the ones that interface to the user, participate in service selection, and perform any off-box signaling responsibilities. Their job is to collect any data-plane related configuration information and inject it into the data-plane setup process through the L2HW switching API.
- Segment Switching Manager (SSM)—This is the main data-plane setup process. The SSM registers for the API events advertised through the L2HW API. The SSM controls the processing of these messages and their dispatch into the Segment and Feature handler FSMs (Finite State Machines). The SSM also takes care of the platform-specific distribution to line cards and takes care of platform events, such as line card failures or insertion events. The SSM includes connection FSMs, per-Segment FSMs, and Feature FSMs. One key aspect of the SSM is that it normalizes the multiple switching engines that might exist in the system into one switching engine from the client's perspective. The SSM clients call into the L2HW once to set up all switching engines. The SSM translates these client requests into multiple calls per class of switching. In the current SAL architecture, two classes of switching exist. The first is software switching, which is defined as the SSS (Subscriber Service Switch) class. The second class is for hardware switching, which is coined the HW class. The segment handlers plug into the SSM via a segment handler vector table. This plug-in is based on segment type and class type.
- Segment Handler Plug-ins—The SSM provides a method for per-segment, per-class plug-ins, and the API for these plug-ins is defined in the segment handler vector table. The segment handler's job is to set up any necessary data-plane state needed to manage the data-plane tables.

and to inject this into the switching path. Since the segment handlers are designed to be pluggable, they are only built where needed. For example, if a platform only supports software switching, the HW-class plug-in will not be included in the makefiles for that platform.

- Feature Handler Plug-ins—The feature handlers are conceptually identical to the segment handlers, with the primary difference being that they are responsible to setting up and enforcing features in the data-plane versus managing the segments themselves. For more information on the feature handler design, refer to EDCS-314353, “SSM Feature Handler Manager Design Specification” and EDCS-310300, “SSM Class Adjacency Feature Handler Library Design Specification”.
- SSS-class segment handlers—The SSS-class segment handlers set up software switching and the use the SSS switch to move packets between the two segments.
- HW-class segment handlers—The HW-class segment handlers are responsible for setting up any platform-independent structures and performing platform-independent HW management. An example of information that is only needed by some hardware platforms is adjacencies. The adjacency information is managed within the HW-class segment handlers because software-switching only platforms don't utilize adjacencies and thus, there are scalability improvements by making adjacency management an optional part of the overall system. Instead of pushing the adjacency management below the platform APIs though, it is managed generically in the HW-class segment handlers. This allows the code to be shared across all platforms that do need adjacencies to be created. The HW-class segment handlers are responsible for advertising this information along with the segment information to the platform-specific APIs. The platform-specific drivers that sit underneath these APIs are responsible for programming the HW-specific ASICs (Application Specific Integrated Circuits).

The SSM also takes care of performing segment- and feature-independent distribution to the line cards. The goal is to run the SSM state machines and plug-ins identically on the LC, which is accomplished by packaging up the information from the L2HW switching APIs and replaying it on the line cards. This allows for almost 100% of the code to be shared between the RP and the LC with little to no distinction about where the segment handlers are running. The platform-specific switching aspects of the LC and RP are typically the only difference in this environment and this is hidden underneath the segment handlers. One important aspect of this design is that the platform APIs called from the HW-class plug-in are called in exactly the same way on both the RP and the LC. Figure 23-33 represents the distribution model for the SSM.

Figure 23-33 SAL Architecture on Distributed Platforms



23.6.2 Typical Segment Events

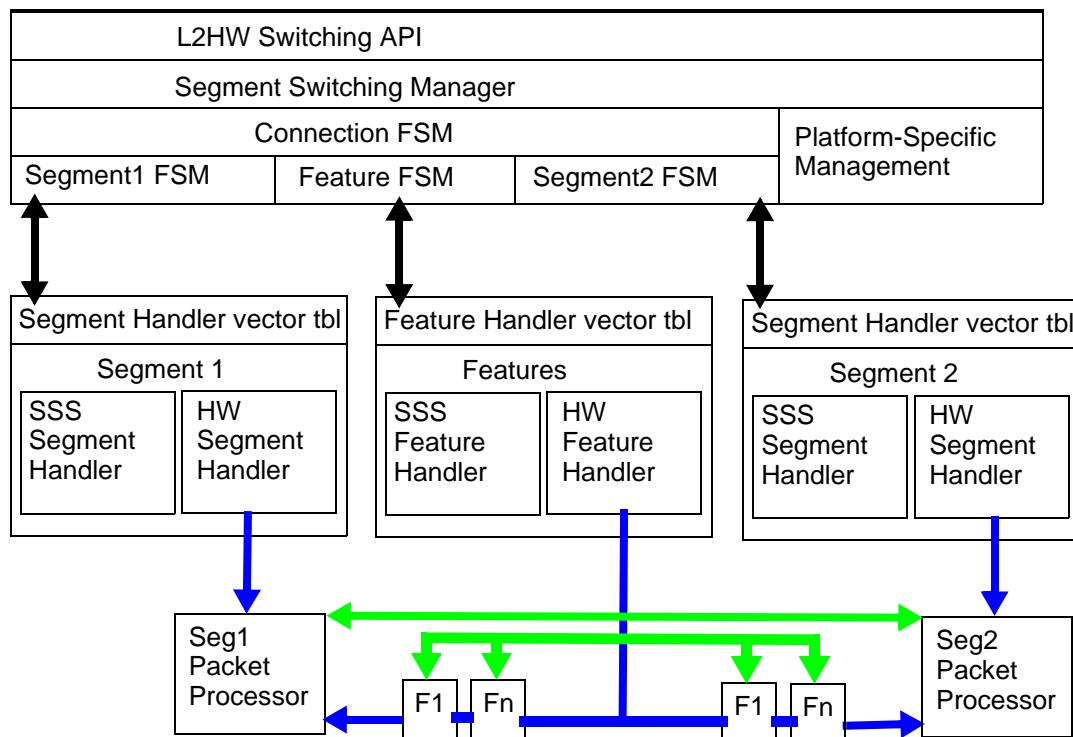
Although each segment has its own APIs, they mostly have a common look and feel. In general, the following events are communicated from each segment:

- Provision—Initial indication about the segment/feature and its data-plane configuration (for example, a provision event sets up the data-plane context for a given segment).
- Update—Notification that something has changed and the contents of the change.
- Bind—Indication that all segments and features on the switch are ready to switch packets.
- Unbind—Indication that the peer segment on the switch is no longer switch packets.

- Unprovision—Removal of the segment/feature from the switch (for example, an unprovision event tears down the data-plane context for a given segment).
- APIs provide accessor functions that allow the platform to obtain the peer segment handle.
- “Pull” accessor APIs that allow the platform to interrogate the control-plane for the data-plane configuration.

The L2VPN Platform API provides the platforms the flexibility to obtain the other segment handle. This “backdoor” method allows the platform to collapse the per-segment data-plane configuration into a single object, as shown in Figure 23-34. This mechanism is also available for the feature APIs. When the platform receives the “bind” for a segment, it can be guaranteed that all of the data-plane state for each segment is ready and available, which allows the platform to coalesce all of the information needed to enforce this service and set up the forwarding tables.

Figure 23-34 How to Obtain the Other Segment Handle



23.6.3 Types of Segment API Functions

The L2VPN Platform API is currently implemented in the 12.0S and 12.2S, and it consists of the following types of segment API functions (see the API reference pages in the *Cisco IOS Reference* for detailed information on these functions):

- AC Segment API
- L2TP Segment API
- AToM Segment API - 12.2S MFI Version
- AToM Segment API - 12.0S AToM Switching Manager
- VFI Segment API

23.6.4 AC Segment API

The AC Segment API includes the following public functions:

- 1 `reg_invoke_ac_switching_provision_circuit()`
- 2 `reg_invoke_ac_switching_update_circuit()`
- 3 `reg_invoke_ac_switching_unprovision_circuit()`
- 4 `reg_invoke_ac_switching_bind_circuit()`
- 5 `reg_invoke_ac_switching_unbind_circuit()`
- 6 `reg_invoke_ac_switching_adjacency_change()`
- 7 `reg_invoke_ac_switching_mqc_policy_change()`
- 8 `reg_invoke_ac_switching_other_segment_updated()`
- 9 `reg_invoke_ac_switching_get_counters()`
- 10 `reg_invoke_ac_switching_send_event()`
- 11 `ac_switching_get_circuit_info()`
- 12 `ac_switching_get_link_raw_adj_object()`
- 13 `ac_switching_get_link_raw_adjacency()`
- 14 `ac_switching_get_segment_info()`
- 15 `ac_switching_get_private_handle()`
- 16 `ac_switching_set_private_handle()`
- 17 `ac_switching_get_other_segtypes()`
- 18 `ac_switching_get_other_seghandle()`
- 19 `ac_switching_push_counters()`

The following header files define the AC segment setup APIs and define how the AC segment information is associated with the peer segment:

- `sys/wan/ac_switching_registry.reg` - Defines the APIs that the attachment circuit HW class segment handlers use to provision the HW data-plane. These APIs include initial provisioning, update, bind, and unprovisioning events.
- `ac_switching_registry.h` - Defines the accessor APIs that allow the platform to pull information about the AC segment and obtain the segment type and handle of the other segment.
- `sys/wan/ac_fr_switching.h` - Frame relay-specific segment configuration.
- `sys/wan/ac_atm_switching.h` - ATM-specific segment configuration.
- `sys/wan/ac_vlan_switching.h` - VLAN-specific segment configuration.
- `sys/wan/ac_ether_switching.h` - Ethernet-specific segment configuration.
- `sys/wan/ac_ppp_switching.h` - PPP-specific segment configuration.
- `sys/wan/ac_hdlc_switching.h` - HDLC-specific segment configuration.

23.6.4.1 Provision an AC

The AC provision API is called whenever an xconnect [point-to-point cross-connection, such as a L2TPv3 (version 3) to AToM connection] is associated with an attachment circuit. Note that at the time when this API is called, the link raw adjacency may be `NULL`. The link raw adjacency may be `NULL` if the provisioning information becomes available before the adjacency has propagated to a line card or has been marked as complete. The platform code must be prepared to handle this

situation and drop packets as appropriate. The platform may use the `private_handle` to associate the HW/platform private context with this attachment circuit. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE. The `ac_switching_info` data structures can be found in the appropriate `ac_*_switching.h` files.

```
void reg_invoke_ac_switching_provision_circuit(boolean *retval,
                                                12hw_segment_type ac_segtypes,
                                                void *ac_seghandle,
                                                void *ac_switching_info,
                                                void **private_handle);
```

23.6.4.2 Update an AC

The AC update API is called whenever an attachment circuit configuration parameter changes. Note that the HW may use the `private_handle` to associate the HW/platform private context with this attachment circuit. If this association was already performed in the provision circuit call, then the `private_handle` contains the pointer set as part of that association. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_update_circuit(boolean *retval,
                                             12hw_segment_type ac_segtypes,
                                             void *ac_seghandle,
                                             void *ac_switching_info,
                                             void *ac_switching_change_info,
                                             void **private_handle);
```

23.6.4.3 Remove an AC

The AC unprovision API is called whenever a user performs an operation that removes the xconnect configuration from an attachment circuit. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_unprovision_circuit(boolean *retval,
                                                 12hw_segment_type ac_segtypes,
                                                 void *ac_seghandle,
                                                 void *ac_switching_info,
                                                 void *private_handle);
```

23.6.4.4 Bind an AC

The AC bind API is called whenever both the AC circuit and the other segment in the xconnect are both provisioned and ready. The platform should use this API as an indication to finish any set up needed to move packets between the two segments. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_bind_circuit(boolean *retval,
                                           12hw_segment_type ac_segtypes,
                                           void *ac_seghandle,
                                           void *ac_switching_info,
                                           void *private_handle);
```

23.6.4.5 Unbind an AC

The AC unbind API is called whenever the other segment in the xconnect has been unprovisioned. The platform should use this API remove any binding between this AC and the other segment. In addition this segment should be marked in such a way that packets do not flow through it for receive or transmit. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_unbind_circuit(boolean *retval,
                                             12hw_segment_type ac_segtpe,
                                             void *ac_seghandle,
                                             void *ac_switching_info,
                                             void *private_handle);
```

23.6.4.6 Adjacency Change Notification

The following API registry function is called whenever the link raw adjacency bound to this AC changes state. The `ac_switching_get_link_raw_adjacency()` accessor function should be used to obtain the adjacency. Note, if the adjacency has been deleted, this accessor function returns NULL. If the adjacency state becomes incomplete, the platform switching code should drop all packets received or transmitted on that interface. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_adjacency_change(boolean *retval,
                                               12hw_segment_type ac_segtpe,
                                               void *ac_seghandle,
                                               void *ac_switching_info,
                                               void *private_handle);
```

23.6.4.7 QoS Policy Updates Notification

The following API registry function is called whenever a MQC (Modular QoS CLI) policy associated with this AC changes in some way. The platform code must find and update the MQC policies associated with this AC. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_mqc_policy_change(boolean *retval,
                                                12hw_segment_type ac_segtpe,
                                                void *ac_seghandle,
                                                void *ac_switching_info,
                                                void *private_handle);
```

23.6.4.8 Other Segment Updates Notification

The following API registry function is called whenever a change occurs on the other segment of the xconnect. In general, this API should not be used by the platform code. Instead, the platform code should register for the updates on the other segment's APIs. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_other_segment_updated(boolean *retval,
                                                   12hw_segment_type ac_segtpe,
                                                   void *ac_seghandle,
                                                   void *ac_switching_info,
                                                   void *private_handle);
```

23.6.4.9 Counter Collection APIs

The following API registry function provides a mechanism to collect counters from the platform for the attachment circuit specified. If this registry returns FALSE, i.e. no one is registered for it, then no work is performed. In addition, if the counter block returns all zeros, then no work is done by the AC segment handler. However, if non-zero counters are returned via this API, the AC segment handler performs the necessary operations to push these counters into the correct IOS data structures. If platform code registers for this registry, it is expected that the platform will clear any counters that it reports back through this API upon returning from this call. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_ac_switching_get_counters(boolean *retval,
                                         12hw_segment_type ac_segtypes,
                                         void *ac_seghandle,
                                         void *ac_switching_info,
                                         hw_switch_counters *counters,
                                         void *private_handle,
                                         boolean reset);
```

23.6.4.10 Platform Send Event Request

The following API registry function is called whenever the platform code needs to send an event to an attachment circuit:

```
void reg_invoke_ac_switching_send_event(uint if_number,
                                         uint circuit_id,
                                         ac_switching_api_events event,
                                         void *msg);
```

23.6.4.11 Retrieve AC Circuit Information

The following API function allows the platform code to retrieve normalized AC circuit information from the attachment circuit segment handle. This saves the platform code from providing its own per-segment decode routine.

```
boolean ac_switching_get_circuit_info(void *ac_seghandle,
                                       ac_switching_circuit_info *ac_info);
```

23.6.4.12 Retrieve LINK_RAW Adjacency Object

These API functions return the adjacency for sending packets into an AC.

For 12.2S:

```
sw_obj_handle ac_switching_get_link_raw_adj_object(
                                         12hw_segment_type ac_segtypes,
                                         void *ac_seghandle);
```

For 12.0S:

```
struct adjacency_ *ac_switching_get_link_raw_adjacency(
                                         12hw_segment_type ac_segtypes,
                                         void *ac_seghandle);
```

23.6.4.13 Retrieve AC Segment Information

The following API function allows the platform code to retrieve the segment configuration information associated with this attachment circuit:

```
l2hw_segment_info *ac_switching_get_segment_info(
    l2hw_segment_type ac_segttype,
    void *ac_seghandle);
```

23.6.4.14 Set and Get the Platform Private Handle for an AC Segment

The following API function allows the platform code to retrieve the private handle:

```
void *ac_switching_get_private_handle(l2hw_segment_type ac_segttype,
                                       void *ac_seghandle);
```

The following API allows the platform code to set the private handle:

```
void ac_switching_set_private_handle(l2hw_segment_type ac_segttype,
                                      void *ac_seghandle,
                                      void *private_handle);
```

23.6.4.15 Retrieve the Peer Segment Handle and Segment Type from the AC Segment Handle

The following API function allows you to retrieve the segment type from the AC segment handle:

```
l2hw_segment_type ac_switching_get_other_segttype(
    l2hw_segment_type ac_segttype,
    void *ac_seghandle);
```

The following API function allows you to retrieve the peer segment handle from the AC segment handle:

```
void *ac_switching_get_other_seghandle(l2hw_segment_type ac_segttype,
                                         void *ac_seghandle);
```

23.6.4.16 Push Counters into the Segment Context

The following API function allows the platform code to “push” hardware counters for a given attachment circuit into the control-plane. This method is used as an alternative, or in conjunction with, the normal polled statistics collection mechanism.

```
boolean ac_switching_push_counters(void *, hw_switch_counters *);
```

23.6.5 L2TP Segment API

The L2TP Segment API includes the following public functions:

- 1 `reg_invoke_l2x_switching_provision_session()`
- 2 `reg_invoke_l2x_switching_update_session()`
- 3 `reg_invoke_l2x_switching_unprovision_session()`
- 4 `reg_invoke_l2x_switching_bind_session()`
- 5 `reg_invoke_l2x_switching_unbind_session()`
- 6 `reg_invoke_l2x_switching_adjacency_change()`
- 7 `reg_invoke_l2x_switching_other_segment_updated()`
- 8 `reg_invoke_l2x_switching_punt_request()`

```
9 l2x_switching_get_counters()
10 l2x_switching_get_pw_adjacency()
11 l2x_switching_get_pw_adj_object()
12 l2x_switching_get_segment_info()
13 l2x_switching_get_private_handle()
14 l2x_switching_set_private_handle()
15 l2x_switching_get_other_segtypes()
16 l2x_switching_get_other_seghandle()
17 l2x_switching_adj_push_counters()
```

The following header files define the APIs to set up the L2TP Session information and how to associate this session with the peer segment.

- `sys/vpn/l2x_switching_registry.reg` - Defines the APIs that the L2TP HW class segment handlers use to provision the HW data-plane. These include initial provisioning, update, bind, and unprovisioning events.
- `l2x_switching_registry.h` - Defines the accessor APIs that allow the platform to pull information about the L2TP segment and obtain the segment type and handle of the other segment.
- `l2x_switching_api.h` - Header file that defines the L2TP-specific segment configuration.

23.6.5.1 Add a L2TP Session

The following L2X [the family of Layer 2 protocols that includes L2TPv2 (version 2), L2TPv3, L2F (Layer Two Forwarding Protocol), and PPTP (Point-to-Point Tunneling Protocol)] provision session API registry function is called after all protocol negotiation has been completed and the information needed for encapsulation and decapsulation is known. The private handle is used to store a pointer or handle to platform-specific data that may be associated with the L2X session during the provision or update calls. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to `FALSE`.

```
void reg_invoke_l2x_switching_provision_session(boolean *retval,
                                                l2hw_segment_type l2x_segtypes,
                                                void *l2x_seghandle,
                                                l2x_switch_info *switching_info,
                                                void **private_handle);
```

23.6.5.2 Update a L2TP Session

The following L2X update session API registry function is called whenever configuration information changes that impacts the handling of data packets. The information that changes is conveyed in the `change_info` data structure. The `l2x_switch_info` is also passed in so that the platform can perform a replace operation if it doesn't support an incremental update. Note, the HW may use the `private_handle` to associate the HW/platform private context with this `l2x` session.

If this association was already performed in the provision call, then the `private_handle` contains the pointer set as part of that association. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_l2x_switching_update_session(boolean *retval,
                                              12hw_segment_type l2x_segtypes,
                                              void *l2x_seghandle,
                                              l2x_switch_info *switching_info,
                                              l2x_switch_change_info *change_info,
                                              void **private_handle)
```

23.6.5.3 Remove a L2TP Session

The following unprovision session API registry function is called whenever the control-plane determines that the L2TP session no longer exists. This can occur as the result of the configuration being removed or because the remote peer is no longer reachable or because it has destroyed the session based on either a normal or abnormal condition. In any case, if this API is called, the platform code must destroy the session and stop switching packets. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_l2x_switching_unprovision_session(boolean *retval,
                                                 12hw_segment_type l2x_segtypes,
                                                 void *l2x_seghandle,
                                                 l2x_switch_info *switching_info,
                                                 void *private_handle);
```

23.6.5.4 Bind a L2TP Session

The following L2X bind API registry function is called whenever the L2X session and the other segment in the scanned are both provisioned and ready. The platform should use this API as an indication to finish any set up needed to move packets between the two segments. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_l2x_switching_bind_session(boolean *retval,
                                            12hw_segment_type l2x_segtypes,
                                            void *l2x_seghandle,
                                            l2x_switch_info *l2x_switching_info,
                                            void *private_handle);
```

23.6.5.5 Unbind a L2TP Session

The following L2X unbind API registry function is called whenever the L2TP session can no longer forward traffic. This may be caused by the L2TP session becoming unavailable or because the other segment can no longer forward traffic. The platform should use this API to remove any binding between this L2X session and the other segment. In addition, this segment should be marked in such a way that packets do not flow through it for receive or transmit. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to FALSE.

```
void reg_invoke_l2x_switching_unbind_session(boolean *retval,
                                              12hw_segment_type l2x_segtypes,
                                              void *l2x_seghandle,
                                              l2x_switch_info *l2x_switching_info,
                                              void private_handle);
```

23.6.5.6 Adjacency Change Notification

The following API registry function is called whenever the pseudowire adjacency bound to this L2X session changes state. The `l2x_switching_get_pw_adjacency()` accessor function should be used to obtain the adjacency. Note, if the adjacency has been deleted, then this accessor function will return `NULL`. If the adjacency state becomes incomplete, the platform switching code should drop all packets received or transmitted on that interface. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to `FALSE`.

```
void reg_invoke_l2x_switching_adjacency_change(boolean *retval,
                                                12hw_segment_type l2x_segtypes,
                                                void *l2x_seghandle,
                                                l2x_switch_info *l2x_switching_info,
                                                void *private_handle);
```

23.6.5.7 Other Segment Updated Notification

The following API registry function is called whenever a change occurs on the other segment of the xconnect. In general, this API should not be used by the platform code. Instead, the platform code should register for the updates on the other segment's APIs. The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to `FALSE`.

```
void reg_invoke_l2x_switching_other_segment_updated(boolean *retval,
                                                    12hw_segment_type l2x_segtypes,
                                                    void *l2x_seghandle,
                                                    l2x_switch_info *l2x_switching_info,
                                                    void *private_handle);
```

23.6.5.8 Start/Stop Punting Packets to Next Switching Level

The following API registry function is called to indicate to the platform that it should start/stop punting all packets to the next switching level (that is, typically to the software path). The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to `FALSE`.

```
static inline void reg_invoke_l2x_switching_punt_request (
    boolean *retval,
    12hw_segment_type l2x_segtypes,
    void *l2x_seghandle,
    l2x_switch_info *l2x_switching_info,
    void *private_handle,
    boolean start_punting)
```

23.6.5.9 Collect Counters on a L2TP Session

The following API function provides a mechanism to collect counters from the platform for the L2X session specified. If this registry returns `FALSE`, i.e. no one is registered for it, then no work is done. In addition, if the counter block returns all zeros, then no work is done by the L2X segment handler. However, if non-zero counters are returned via this API, the L2X segment handler performs the necessary operations to push these counters into the correct IOS data structures. If platform code registers for this registry, it's expected that the platform will clear any counters that it reports back through this API upon returning from this call. The exceptions to this are any counters that represent state, such as sequence numbers (those counters must not be reset). The `retval` parameter is used to indicate a failure. If a failure occurs, set this value to `FALSE`.

```
void l2x_switching_get_counters(boolean *retval,
                                 12hw_segment_type l2x_segtypes,
                                 void *l2x_seghandle,
```

```
l2x_switch_info *switching_info,  
hw_switch_counters *counters,  
void *private_handle);
```

23.6.5.10 Retrieve LINK_PW_IP Adjacency Object

The following API functions return the adjacency for sending packets into a L2TP tunnel.

For 12.0S:

```
struct adjacency_ * l2x_switching_get_pw_adjacency(  
                                         12hw_segment_type l2x_segtypes,  
                                         void *l2x_seghandle);
```

For 12.2S:

```
sw_obj_handle l2x_switching_get_pw_adj_object(12hw_segment_type l2x_segtypes,  
                                              void *l2x_seghandle);
```

23.6.5.11 Return L2TP Segment Configuration

The following API function allows the platform code to retrieve the segment configuration information associated with a L2TP segment handle:

```
12hw_segment_info *l2x_switching_get_segment_info(  
                                         12hw_segment_type l2x_segtypes,  
                                         void *l2x_seghandle);
```

23.6.5.12 Set and Get the Platform Private Handle for a L2TP Segment

The following API function allows the platform code to retrieve the platform private handle associated with a L2TP segment handle:

```
void *l2x_switching_get_private_handle(12hw_segment_type l2x_segtypes,  
                                         void *l2x_seghandle);
```

The following API function allows the platform code to set the platform private handle associated with a L2TP segment handle:

```
void l2x_switching_set_private_handle(12hw_segment_type l2x_segtypes,  
                                         void *l2x_seghandle,  
                                         void *private_handle);
```

23.6.5.13 Retrieve the Peer Segment Handle and Segment Type from the L2TP Segment Handle

The following API function allows the platform code to retrieve the peer segment type associated with a L2TP segment handle:

```
12hw_segment_type l2x_switching_get_other_segtypes(  
                                         12hw_segment_type l2x_segtypes,  
                                         void *l2x_seghandle);
```

The following API function allows the platform code to retrieve the peer segment handle associated with a L2TP segment handle:

```
void *l2x_switching_get_other_seghandle(12hw_segment_type l2x_segtypes,  
                                         void *l2x_seghandle);
```

23.6.5.14 Return L2TP Segment Configuration

The following API function allows the platform code to push hardware counters into adjacency context, for the next periodic counter collection, to the RP:

```
boolean l2x_switching_adj_push_counters(void *l2x_seghandle,
                                         hw_switch_counters *counters);
```

23.6.6 AToM Segment API - 12.2S MFI Version

The AToM Segment API - 12.2S MFI Version includes the following public functions:

- 1 atom_disp_get_seg_class()
- 2 atom_disp_has_controlword()
- 3 atom_disp_has_sequencing()
- 4 atom_imp_has_controlword()
- 5 atom_imp_has_sequencing()
- 6 atom_imp_oce_get_loadbalance_hash()
- 7 ssm_get_atom_imp_oce()

All of the API functions to set up and manage label paths and associate these labels with the peer segments for MFI (MPLS Forwarding Infrastructure) are defined in the following header file:

sys/oce/hw_api.h

23.6.6.1 Retrieve Peer Segment Information

The following API function is used to get the peer segment handle and peer segment type when provided with the AToM disposition (coming out of the MPLS core toward the attachment circuit) chain element:

```
sw_obj_handle atom_disp_get_seg_class(sw_mgr_segment_classes_t ssm_class,
                                       sw_obj_handle disp_oce,
                                       l2hw_segment_type *segtype);
```

23.6.6.2 Determine If Disposition/Imposition Has Controlword/Sequencing Processing

The following API functions are used to determine if the disposition or imposition (going into the MPLS core away from the attachment circuit) has controlword or sequencing processing enabled:

- boolean atom_disp_has_controlword(sw_obj_handle atom_disp);
- boolean atom_disp_has_sequencing(sw_obj_handle atom_disp);
- boolean atom_imp_has_controlword(sw_obj_handle atom_imp);
- boolean atom_imp_has_sequencing(sw_obj_handle atom_imp);

23.6.6.3 Retrieve the loadbalance Index

The following API function is used to retrieve the loadbalance index from the hash of the virtual circuit label:

```
sw_obj_handle
atom_imp_oce_get_loadbalance_hash (sw_obj_handle chain_head,
                                    sw_obj_type head_type,
                                    sw_obj_handle lbl_oce,
                                    sw_obj_type *type);
```

23.6.6.4 Retrieve the AToM Imposition OCE

The following API function is used to retrieve the AToM imposition OCE (Output Chain Element) from the SSM segment class handle:

```
sw_obj_handle ssm_get_atom_imp_oce(sw_obj_handle seg_class);
```

23.6.6.5 Program a tag-rewrites

Refer to EDCS-222041 for more details on how label objects are managed. The HW API vectors for the SW_OBJ_LABEL are used to notify the platform of create, delete, and modify events on the imposition and disposition VC label objects. These vectors are:

- sw_fwding_obj_created_t

Create SSS Object Notification—Notification of create events for the SW_OBJ_SSS object is delivered through the HW API vector table associated with the SW_OBJ_SSS object. This vector is set using hw_api_set_common_vectors().

- sw_fwding_obj_deleted_t

Delete SSS Object Notification—Notification of delete events for the SW_OBJ_SSS object is delivered through the HW API vector table associated with the SW_OBJ_SSS object. This vector is set using hw_api_set_common_vectors().

- sw_fwding_obj_modified_t

Disposition Bind and Unbind Event Notification—Notification from the MFI that the SSS object (SW_OBJ_SSS) has been bound or unbound from the peer segment. Use the sw_fwding_obj_modified_t vector of the SSS sw_obj vector table.

23.6.7 AToM Segment API - 12.0S AToM Switching Manager

The AToM Segment API - 12.0S includes the following public functions:

- 1 atom_smgr_get_disp_rew_by_atom_seg_class()
- 2 atom_smgr_get_hw_private_handle()
- 3 atom_smgr_get_imp_rew_by_atom_seg_class()
- 4 atom_smgr_get_other_adj_seghandle()
- 5 atom_smgr_get_other_adj_segtypes()
- 6 atom_smgr_get_other_seghandle_by_atom_seg_class()
- 7 atom_smgr_get_other_segtypes_by_atom_seg_class()
- 8 atom_smgr_get_sss_segid_by_atom_adj_segclass()
- 9 atom_smgr_set_hw_private_handle()
- 10 reg_invoke_delete_hw_rewrite()
- 11 reg_invoke_update_hw_rewrite()

The following header files define the APIs to set up the tag-switching rewrites and to associate these rewrites with the peer segment:

- sys/tagsw/tagsw_registry.reg
- sys/atom/atom_smgr.h

23.6.7.1 Program a tag-rewrites

The following API registry functions are used manage the imposition and disposition tag rewrites associated with the AToM VC:

- `void reg_invoke_update_hw_rewrite(tag_rewrite *rewrite);`
- `void reg_invoke_delete_hw_rewrite(tag_rewrite *rewrite);`

23.6.7.2 Retrieve Peer Segment Information

The following API functions are used to retrieve the peer segment handle and segment type:

- `void *atom_smgr_get_other_adj_seghandle(tag_rewrite *dispose_rewrite);`
- `l2hw_segment_type atom_smgr_get_other_adj_segttype(tag_rewrite *dispose_rewrite);`
- `void *atom_smgr_get_other_seghandle_by_atom_seg_class(void *seg_class);`
- `l2hw_segment_type atom_smgr_get_other_segttype_by_atom_seg_class(void *seg_class);`

23.6.7.3 Retrieve Imposition Rewrite from AToM Segment Handle

The following API function is used to retrieve the imposition rewrite when given the AToM segment class handle:

```
tag_rewrite *atom_smgr_get_imp_rew_by_atom_seg_class(void *seg_class);
```

23.6.7.4 Retrieve Disposition Rewrite from AToM Segment Handle

The following API function is used to retrieve the disposition rewrite when given the AToM segment class handle:

```
tag_rewrite *atom_smgr_get_disp_rew_by_atom_seg_class(void *seg_class);
```

23.6.7.5 Set AToM Segment Private Platform Context

The following API function allows the platform code to set/associate the platform handle with the AToM segment handle:

```
void atom_smgr_set_hw_private_handle(void *atom_seghandle,  
                                      void *private_handle);
```

23.6.7.6 Get AToM Segment Private Platform Context

The following API function allows the platform code to retrieve the platform handle when given the AToM segment handle:

```
void *atom_smgr_get_hw_private_handle(void *atom_seghandle);
```

23.6.7.7 Retrieve SSM Segment ID from AToM Segment

The following API function allows the platform code to retrieve the globally unique SSM segment ID associated with this AToM segment:

```
uint atom_smgr_get_sss_segid_by_atom_adj_segclass(void *seg_class);
```

23.6.8 VFI Segment API

The VFI (Virtual Forwarding Instance) Segment API includes the following public functions:

- 1 `reg_invoke_vfi_switching_provision_circuit()`
- 2 `reg_invoke_vfi_switching_unprovision_circuit()`
- 3 `reg_invoke_vfi_switching_update_circuit()`
- 4 `reg_invoke_vfi_switching_bind_circuit()`
- 5 `reg_invoke_vfi_switching_unbind_circuit()`
- 6 `reg_invoke_vfi_compute_platform_index()`
- 7 `vfi_switching_get_segment_info()`
- 8 `vfi_switching_get_private_handle()`
- 9 `vfi_switching_set_private_handle()`
- 10 `vfi_switching_get_other_segtypes()`
- 11 `vfi_switching_get_other_seghandle()`
- 12 `vfi_segment_list_walk_req()`

The following header files define the VFI segment setup APIs and define how the VFI segment information is associated with the peer segment:

- `sys/vfi/vfi_switching_registry.reg` - Defines the APIs that the VFI HW class segment handler uses to program the VPLS hardware data-plane. The API functions deal with VFI segment provision, unprovision, bind, unbind, and update events.
- `sys/vfi/vfi_switching_registry.h` - Defines the accessor APIs that allow the platform to retrieve information about the VFI segment and obtain the segment type and handle of the other segment.

The generic platform registry functions are defined as follows: the platform hardware abstraction layer “`reg_add`” to these functions, and they are called by the VFI segment handler when certain data-plane event occurs.

23.6.8.1 Provision a VFI Segment

The following API registry function provisions a VFI segment. The private handle is used to store a pointer or handle for platform-specific data that may be associated with the VFI segment during the provision or update calls.

```
static inline void reg_invoke_vfi_switching_provision_circuit (
    boolean *retval,
    l2hw_segment_type vfi_segtypes,
    void *vfi_seghandle,
    vfi_segment_info *vfi_switching_info,
    void **private_handle);
```

23.6.8.2 Unprovision a VFI Segment

The following API registry function unprovisions a VFI segment. The platform hardware abstraction layer must remove data structures associated with the VFI segment and stop packet forwarding when this API is called.

```
void reg_invoke_vfi_switching_unprovision_circuit (
    boolean *retval,
    l2hw_segment_type vfi_segtypes,
    void *vfi_seghandle,
    vfi_segment_info *vfi_switching_info,
    void *private_handle);
```

23.6.8.3 Update a VFI Segment

The following API registry function updates a VFI segment. The segment information being changed is conveyed through the `change_info` structure.

```
void reg_invoke_vfi_switching_update_circuit (
    boolean *retval,
    l2hw_segment_type vfi_segtypes,
    void *vfi_seghandle,
    vfi_segment_info *vfi_switching_info,
    void *vfi_switching_change_info,
    void **private_handle);
```

23.6.8.4 Bind a VFI Segment

The following API registry function binds a VFI segment. This function is called when both segments of the switch are provisioned for and ready for forwarding packets.

```
static inline void reg_invoke_vfi_switching_bind_circuit (
    boolean *retval,
    l2hw_segment_type vfi_segtypes,
    void *vfi_seghandle,
    vfi_segment_info *vfi_switching_info,
    void *private_handle);
```

23.6.8.5 Unbind a VFI Segment

The following API registry function unbinds a VFI segment. This function is called when the segment on the other side of the switch is unprovisioned.

```
static inline void reg_invoke_vfi_switching_unbind_circuit (
    boolean *retval,
    l2hw_segment_type vfi_segtypes,
    void *vfi_seghandle,
    vfi_segment_info *vfi_switching_info,
    void *private_handle);
```

23.6.8.6 Compute and Reserve a Platform Resource

Some platforms (for example, 7600) need to reserve hardware resources for creating the PW (pseudowire) - VFI switch, and need to do so before provisioning the VFI segment. Therefore, the following registry function is defined and it needs to be called before the provisioning call by the

VFI manager (as opposed to the VFI segment handler). These registry functions are defined in `sys/vfi/vfi_registry.reg`. For platforms that do not need to do this computation, it returns a default value of zero.

```
ulong reg_invoke_vfi_compute_platform_index(ulong vlan_id, ulong remote_id);
```

The accessor functions are *not* defined as registry functions. They are called by the platform hardware abstraction layer to retrieve switching information from the VFI segment or the segment on the other side of the switch, for example, AC or PW.

23.6.8.7 Retrieve VFI Segment Information from a VFI Segment Handle

The following API function is used to retrieve VFI segment information from a VFI segment handle:

```
l2hw_segment_info *vfi_switching_get_segment_info(void *vfi_seghandle);
```

23.6.8.8 Retrieve the Platform Private Handle from a VFI Segment Handle

The following API function is used to retrieve the platform private handle from a VFI segment handle:

```
void *vfi_switching_get_private_handle(void *vfi_seghandle);
```

23.6.8.9 Set the Platform Private Handle into a VFI Segment Handle

The following API function is used to set the platform private handle in a VFI segment handle:

```
void vfi_switching_set_private_handle(void *vfi_seghandle,
                                      void *private_handle);
```

23.6.8.10 Retrieve the Segment Type of the Other Segment from a VFI Segment Handle

The following API function is used to get the segment type of the other segment from a VFI segment handle:

```
l2hw_segment_type vfi_switching_get_other_segttype (void *vfi_seghandle);
```

23.6.8.11 Retrieve the Segment Handle of the Other Segment from a VFI Segment Handle

The following API function is used to get the segment handle of the other segment from a VFI segment handle:

```
void *vfi_switching_get_other_seghandle (void *vfi_seghandle)
```

23.6.8.12 Iterate Through all Switches Belonging to a Given VFI

The following API function queues a request to SSM's segment handler queue for a VFI-specific segment update action, passing along the caller-supplied `action_func` and `action_data` that will be applied to each segment of the VFI when the request is processed by SSM Manager.

The initial cursor value must be set to zero.

```
boolean vfi_segment_list_walk_req(vfi_id_t vfi_id,
                                   vfi_class_action_func action_func,
                                   vfi_class_action_type action_type,
                                   void *private_action_data,
                                   vfi_class_free_action_data_func free_action_data);
```

23.6.9 HW API Usage Examples

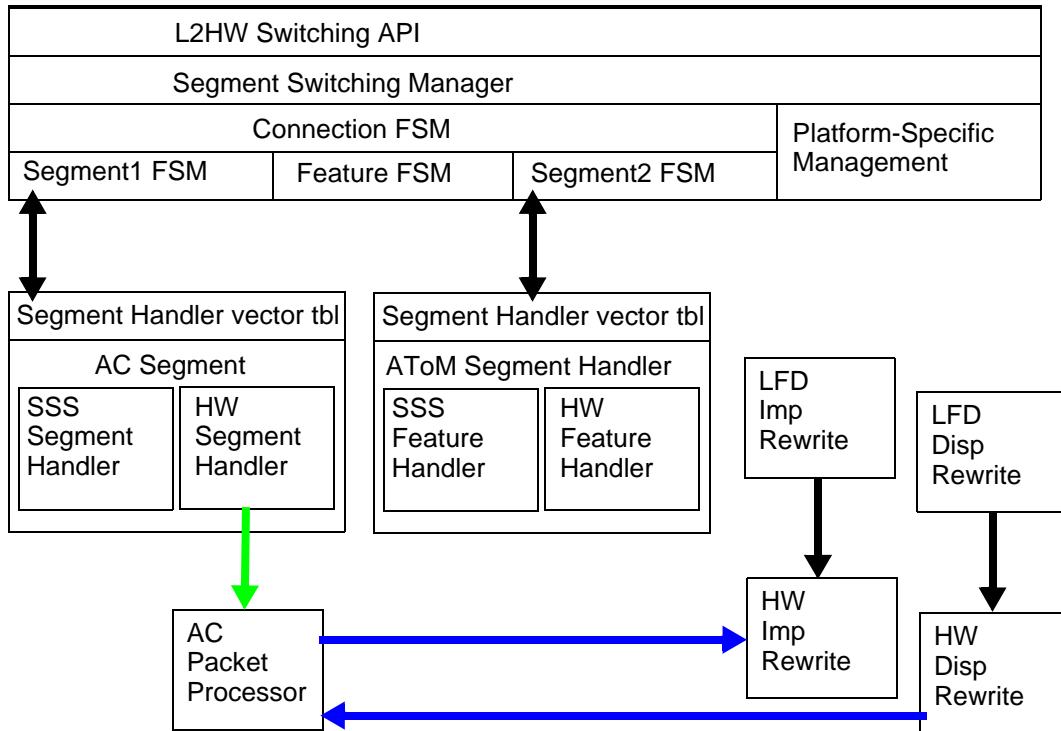
The following sections describe the various usage scenarios for the HW APIs defined above.

23.6.9.1 AC to AToM HW API Overview

Figure 23-35 depicts the API relationship when the service definition is AC to MPLS. Note, this picture is only relevant to the 12.2S, post MFI code base. The SSM provides the control-plane binding between these segments. The characteristics of these APIs are as follows:

- AC APIs are defined in `sys/wan/ac_switching_registry.*`
- MPLS HW APIs are defined in `sys/oce/hw_api.h`
- Accessor functions to “link” AC to labels OCEs and vice-versa are:
 - `ac_switching_get_other_segtypes()`
 - `ac_switching_get_other_seghandle()`
 - `ssm_get_atom_imp_oce()`—Used to map the handle returned from `ac_switching_get_other_seghandle()` to the imposition OCE chain.
 - `atom_disp_get_seg_class()`—Used to map the disposition OCE chain to the AC segment handle.
 - `ac_switching_get_segment_info()`—Given the AC segment handle from the disposition OCE, returns the AC segment information.
- `ac_switching_get_link_raw_adjacency()`—Returns the `LINK_RAW` adjacency. Use this API in 12.0S, non-CSSR (CEF Scalability and Selective Rewrite)/MFI versions.
- `ac_switching_get_link_raw_adj_object()`—Returns the `sw_obj_handle` associated with the `LINK_RAW` adjacency. Use this API in 12.2S, post CSSR/MFI versions.
- Currently, there is no way to go from the imposition OCE to the AC seghandle. In addition, currently there is no way to go from the AC seghandle to the disposition OCE. These enhancements are under investigation.

Figure 23-35 AToM to HW API Diagram

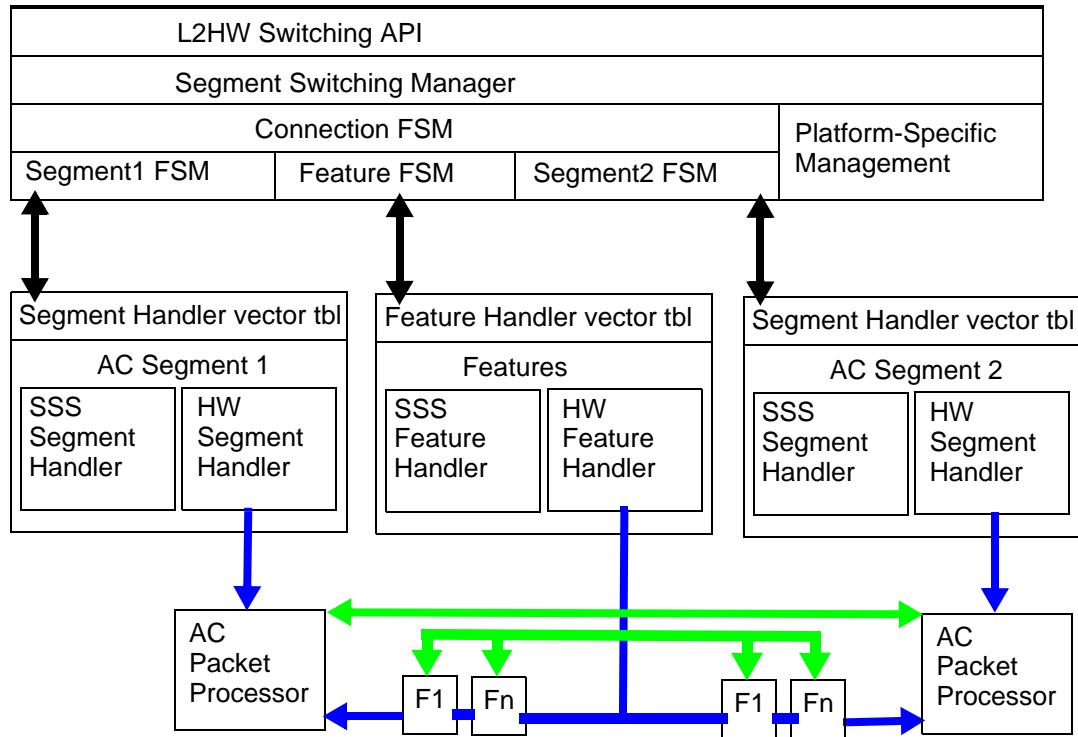


23.6.9.2 AC to AC (Local-switching) HW API Overview

Figure 23-36 depicts the API relationship when the service definition is AC to AC. The SSM provides the control-plane binding between these segments. The characteristics of these APIs are as follows:

- AC APIs are defined in `sys/wan/ac_switching_registry.*`
- Both segments will be set up via the APIs in `ac_switching_registry.*`
- Accessor functions to “link” AC to AC:
 - `ac_switching_get_other_segtypes()`
 - `ac_switching_get_other_seghandle()`
 - `ac_switching_get_segment_info()`—Given the AC segment handle from the above API functions, returns the AC segment information.
- `ac_switching_get_link_raw_adjacency()`—Returns the `LINK_RAW` adjacency. Use this API in 12.0S, non-CSSR/MFI versions.
- `ac_switching_get_link_raw_adj_object()`—Returns the `sw_obj_handle` associated with the `LINK_RAW` adjacency. Use this API in 12.2S, post CSSR/MFI versions.

Figure 23-36 AC to AC HW API Diagram

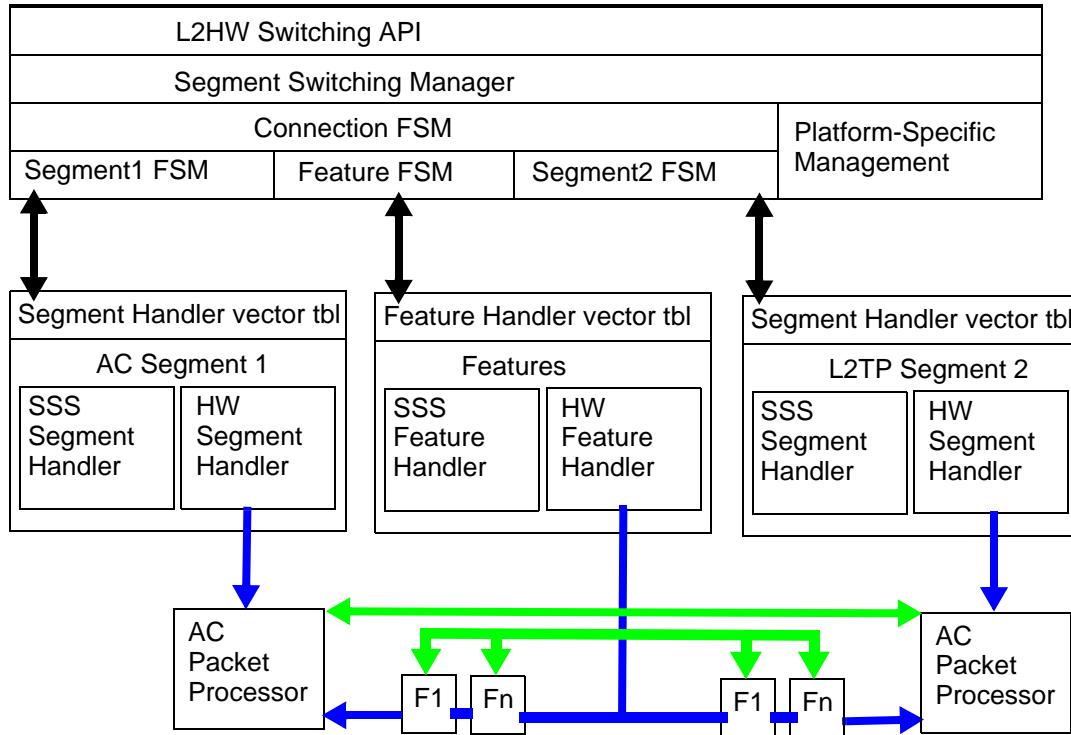


23.6.9.3 AC to L2TP HW API Overview

Figure 23-37 depicts the API relationship when the service definition is AC to L2TP. The SSM provides the control-plane binding between these segments. The characteristics of these APIs are as follows:

- AC APIs are defined in `sys/wan/ac_switching_registry.*`
- L2TP APIs are defined in `sys/vpn/l2x_switching_registry.*`
- Accessor functions to “link” AC to L2TP segment:
 - `ac_switching_get_other_segtypes()`
 - `ac_switching_get_other_seghandle()`
 - `l2x_switching_get_segment_info()`—Given the L2TP segment handle obtained from `ac_switching_get_other_seghandle()`, this API function will return the L2TP segment configuration.
- Accessor functions to “link” L2TP segment to the AC segment:
 - `l2x_switching_get_other_segtypes()`
 - `l2x_switching_get_other_seghandle()`
 - `ac_switching_get_segment_info()`—Given the AC segment handle obtained from the `l2x_switching_get_other_seghandle()` API, returns the AC segment information.
- `ac_switching_get_link_raw_adjacency()`—Returns the `LINK_RAW` adjacency. Use this API in 12.0S, non-CSSR/MFI versions.

- `ac_switching_get_link_raw_adj_object()`—Returns the `sw_obj_handle` associated with the `LINK_RAW` adjacency. Use this API in 12.2S, post-CSSR/MFI versions.
- `l2x_switching_get_pw_adjacency()`—Returns the `LINK_PW_IP` adjacency associated with sending packets into the L2TP pseudowire. Use this API in 12.0S, pre-CSSR/MFI versions.
- `l2x_switching_get_pw_adj_object()`—Returns the `sw_obj_handle` associated with the `LINK_PW_IP` adjacency. Use this API in 12.2S, post-CSSR/MFI versions.

Figure 23-37 AC to L2TP HW API Diagram

23.6.9.4 VFI HW API Overview

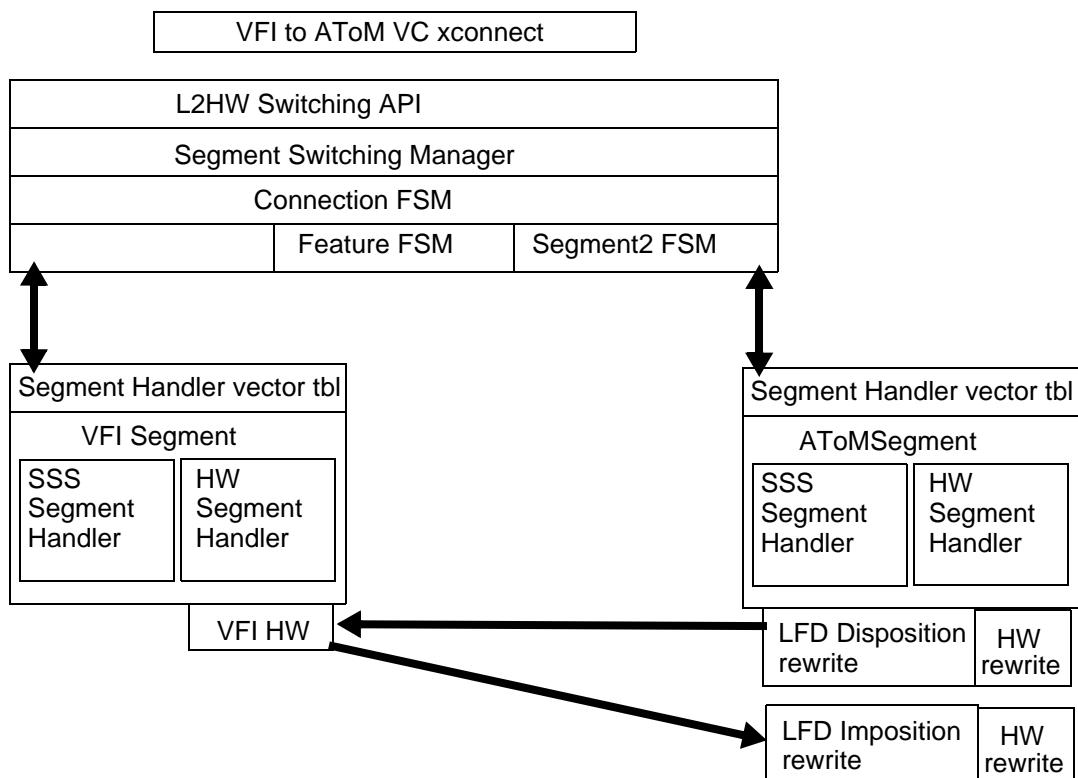
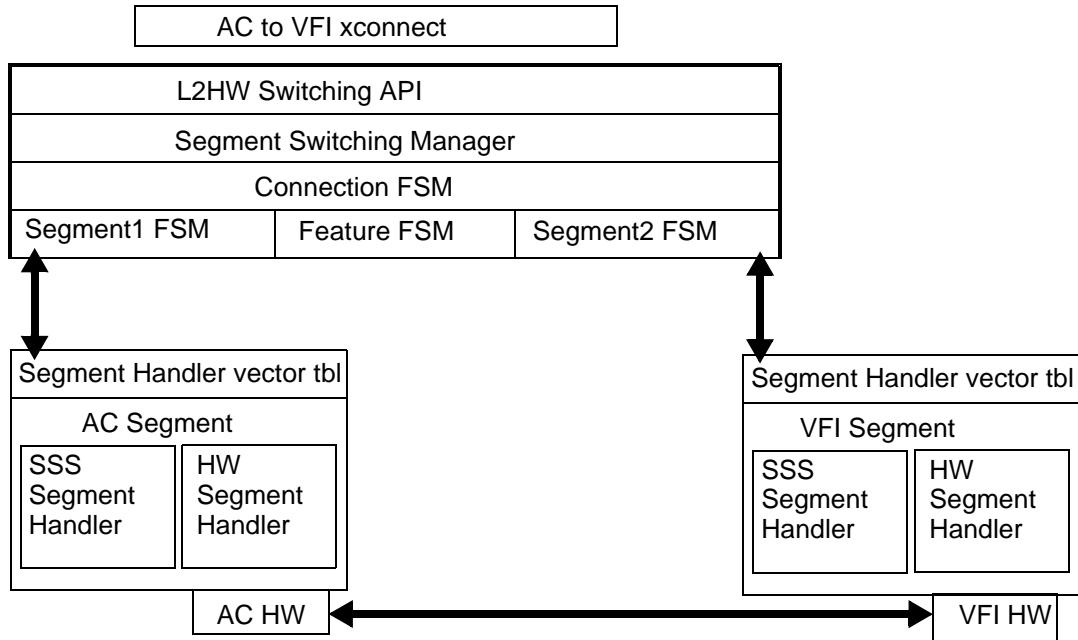
VFI platform APIs consist of registry functions for the platform hardware abstraction layer to respond to data-plane events, and accessor functions to retrieve information of the VFI segment and the segment on the other side of the switch.

Figure 23-38 depicts the API relationship when the service definition is AC to VFI and PW to VFI. The SSM provides the control-plane binding between these segments. The characteristics of these APIs are as follows:

- AC APIs are defined in `sys/wan/ac_switching_registry.*`
- MPLS HW APIs are defined in `sys/oce/hw_api.h`
- VFI APIs are defined in `sys/vfi/vfi_switching_registry.*`
- Accessor functions to “link” AC to VFI segment:
 - `ac_switching_get_other_segtypes()`
 - `ac_switching_get_other_seghandle()`

- `vfi_switching_get_segment_info()`—Given the VFI segment handle obtained from `ac_switching_get_other_seghandle()`, this API function returns the VFI segment configuration.
- Accessor functions to “link” VFI segment to the AC segment:
 - `vfi_switching_get_other_segtypes()`
 - `vfi_switching_get_other_seghandle()`
 - `vfi_switching_get_segment_info()`—Given the AC segment handle obtained from the `vfi_switching_get_other_seghandle()` API, this API function returns the AC segment information.
- Accessor functions to “link” VFI to labels OCEs and vice-versa are:
 - `vfi_switching_get_other_segtypes()`
 - `vfi_switching_get_other_seghandle()`
 - `ssm_get_atom_imp_oce()`—Used to map the handle returned from `vfi_switching_get_other_seghandle()` to the imposition OCE chain.
 - `atom_disp_get_seg_class()`—Used to map the disposition OCE chain to the VFI segment handle.
 - `vfi_switching_get_segment_info()`—Given the VFI segment handle from the disposition OCE, returns the VFI segment information.
- Currently, there is no way to go from the imposition OCE to the VFI seghandle. In addition, currently there is no way to go from the VFI seghandle to the disposition OCE. These enhancements are under investigation.

Figure 23-38 AC to VFI and PW to VFI HW Diagram



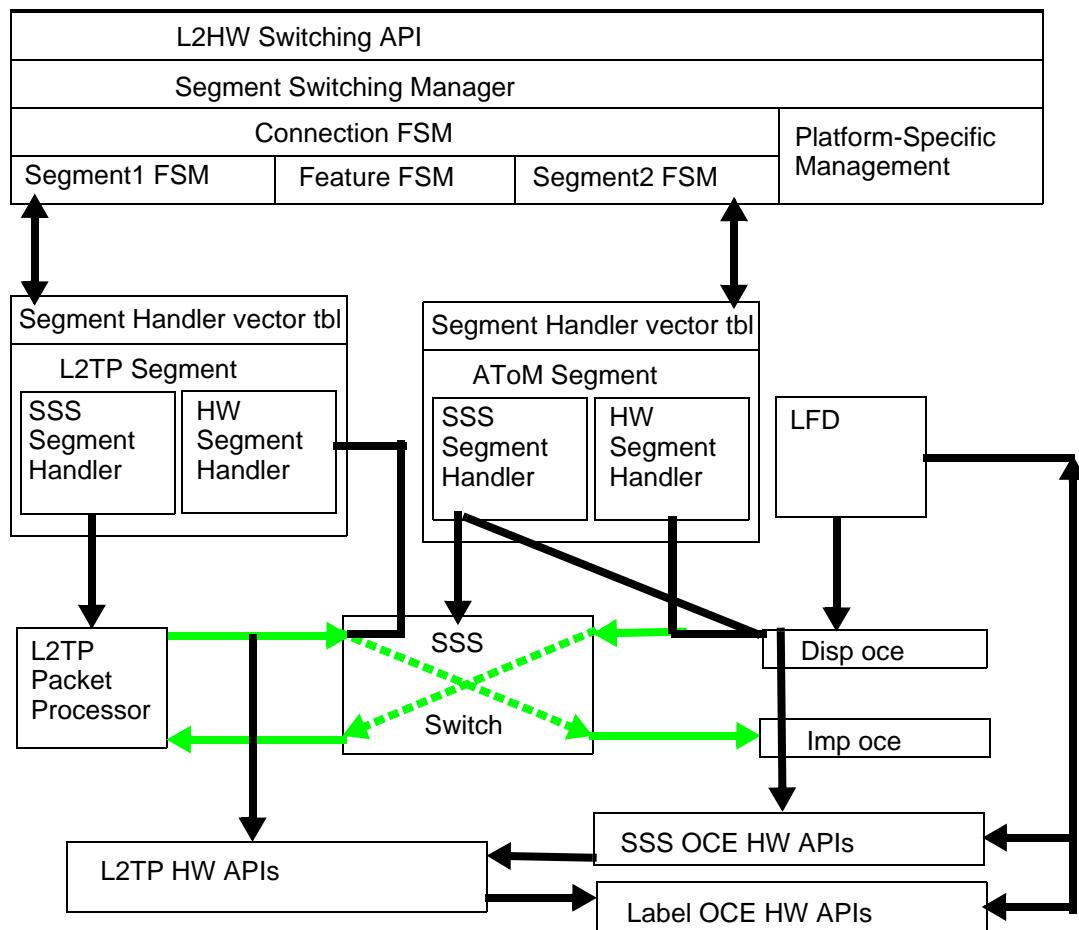
23.6.9.5 Tunnel Stitching HW API Overview

The same HW APIs defined in the previous sections will be used to set up PW-PW xconnects.

Figure 23-39 illustrates the data-plane setup and how the HW APIs are managed from the control-plane. The three tunnel switching cases are as follows:

- MPLS-MPLS. The imposition chain is advertised using the standard label OCE HW APIs. The disposition OCE chain is also advertised using the standard label OCE HW APIs. At the end of the disposition OCE, there exists a SSS SW Object. From the SSS SW Object, the `atom_disp_get_seg_class()` API can be used to retrieve the handle of the peer segment. Using this handle, the platform can call into the `ssm_get_atom_imp_oce()` API to retrieve the imposition label OCE chain.
- L2TP-L2TP. The APIs defined in `sys/vpn/l2x_switching_registry.[h/reg]` will be used to set up and maintain individual L2TP sessions. The main events defined in this API are provision, update, bind, unbind, and unprovision. The platform may use the `l2x_switching_get_other_segtypes()`, `l2x_switching_get_other_seghandle()`, and `l2x_switching_get_segment_info()` accessor APIs to obtain information about the second segment during these main events.
- MPLS-L2TP. In the MPLS-L2TP case, a L2TP segment must be bound to an imposition OCE chain. The VC label disposition OCE chain must be bound to the L2TP segment. The same APIs described above will be used to perform this association. The standard MPLS HW APIs will be used to describe the OCE chains to the HW. The disposition OCE chain will terminate in a SSS SW Object. From the SSS SW Object, the `atom_disp_get_seg_class()` HW API may be used to retrieve the handle of the L2TP segment. From this handle, the `l2x_switching_get_segment_info()` accessor API may be used to retrieve the data-plane configuration information that describes the L2TP segment. The L2TP HW APIs defined in `l2x_switching_registry.*` will be used to set up and manage the L2TP segment. The `l2x_switching_get_other_segtypes()` and `l2x_switching_get_other_seghandle()` APIs can be used to retrieve the AToM Segment handle corresponding to the imposition OCE chain. From this handle, the `ssm_get_atom_imp_oce()` API can be used to retrieve the imposition OCE chain.

Figure 23-39 Tunnel Switching HW Diagram



High Availability (HA)

This chapter is under development. (June, 2001). For the latest information about the High Availability Program, see http://wwwin-eng.cisco.com/Eng/IOS/IOS_Tech/High_Avail_Pgm/

Only three platforms are currently delivering RPR+ and building from 12.2S:- 7500, GSR and ESR. RPR+ code is not necessarily functional in all trains (T and E for example). Many platforms will continue to support EHSA. For information about EHSA, see “Enhanced High System Availability (EHSA)” in the System Initialization chapter.

Note Cisco IOS high availability development questions can be directed to the ios-ha-dev@cisco.com alias.

Added new section “Private Configuration” to section 24.4.2.3, “IOS Infrastructure Components,” to explain the mechanism by which the private configuration is sync’ed with the Standby RP.
(February 2008)

24.1 Introduction

This chapter introduces some of the hardware, firmware, and software features that collectively provide a router’s HA.

- Platform Hardware Model

The backbone of HA is provided by router processor redundancy, or RPR; two router processors are running IOS. One of these RPs assumes control of the network interfaces and provides service (this one is known as the “Active” RP) while the other RP operates as a fully initialized mirrored copy waiting to take control should a fault occur or a manual switchover request be made on the Active. This “RP-in-waiting” is known as the “Standby” RP.

- SSO Software Architectural Model

IOS HA Stateful Switchover (SSO) combines Route Processor redundancy with IOS software enhancements to provide containment of software and hardware faults, to allow non-disruptive (or minimally-disruptive) switchover from the Active RP to a redundant “hot” Standby RP to provide an increase in network availability.

- Redundancy Facility

The Redundancy Facility (RF) is a portable framework that provides synchronization and switchover coordination between redundant processors running IOS. The RF infrastructure provides a series of client services as well as functions for control and monitoring of system redundancy. RF provides notification of transition events which occur on both processors to its clients. RF is key to the operation of a redundant system.

- Target Platform Architectures

The initial target platform set, 7500, 10000, GSR, meet the requirements to support the development of IOS HA. However, this document also mentions the 6400 and Harley / 6100 NI-2 platforms, which have been included in the HA development program plan.

24.1.1 Terms

Active

See Active RP.

Active RP

The RP which controls the system, provides network services, runs the routing protocols, and presents the system management interface.

ATM

Asynchronous Transfer Mode.

ARP

Address Resolution Protocol.

Availability

Within this document, the term availability refers to the percentage of time a software component is ready and able to provide its services; especially regarding how quickly it can continue to provide service after a switchover.

CEF

Cisco Express Forwarding.

CF

Checkpointing Facility.

Checkpointing

Refers to the saving (that is, synchronization) of client-specific state data which will be transferred to a peer client on a remote RP. Once a valid Active to Standby peer client checkpointing session is established, the checkpointed state data will be guaranteed to be delivered to the remote peer client at most once, in order, and without corruption.

Cutover

Deprecated by the term “switchover.”

dCEF

Distributed CEF.

DLCI

Data Link Connection Identifier.

DPM

Defects Per Million. The concept of “defects per million,” or DPM for short, was developed by a major Cisco carrier customer and used to quantify the impact of failures on network availability. The concept has been adopted for use within Cisco for the HA project.

DPM as applied to PVCs may be defined as the number of outage minutes per million connection minutes in service.

A sample mapping of DPM to availability:

10 DPM = 99.999% availability

50 DPM = 99.995% availability

100 DPM = 99.99% availability.

In evaluating aggregate DPM, operational experience favors a larger number of small-DPM outages over a smaller number of large-DPM outages. This suggests improvement efforts should concentrate on containing faults so that a smaller number of subscribers are affected and also on reducing MTTR.

ELMI

Enhanced LMI

FP

Forwarding Processor. On the c10000, the PXF pipeline and associated electronics on the Performance Routing Engine. On the 7500, a VIP card. On the GSR, a GLC card.

HA

High Availability.

Hitless Software Upgrade

The goal of Hitless Software Upgrade is to enable upgrading or downgrading the running IOS code image on a router with no loss of the sessions. It is targeted at reducing the DPM associated with planned upgrades. Full Hitless Software Upgrade of IOS on the RP requires dual RPs in order to operate, whereas Hitless Software Upgrade of Line Card software can only be achieved with redundant line cards.

HSU

See Hitless Software Upgrade.

HA-aware

A feature or protocol is said to be “HA aware” if it has been designed or modified to support, either completely or partially, undisturbed function through an RP switchover.

HA-unaware

A feature or protocol is said to be “HA unaware” if it has not been designed or modified for SSO.

ILMI

Integrated Local Management Interface.

IS-IS

Intermediate System-to-Intermediate System interior gateway routing protocol.

LC

Line card.

LMI

Local Management Interface.

Maintenance Mode

A system state in which both RPs are present but logically separate and not running the HA algorithms. The operator may place the system in split mode for maintenance or software upgrade (that is, HSU).

MTBF

Mean-Time-Between-Failures is the average expected time between failures of a product, assuming the product goes through repeated periods of failure and repair. MTBF applies when a product is in its steady-state random-failure life stage (that is, after the infant mortality and before the wear-out periods), and is equal to the reciprocal of the corresponding constant failure rate. A useful interpretation of MTBF is that within the period of MTBF, 63% of the product population is expected to have failed at least once. MTBF is typically used to describe individual circuit cards.

MTTR

Mean-Time-To-Repair is the average expected time to restore a product from a failure. It represents the period that the product is out of service because of a failure, and is measured from the time that the failure occurs until the time the product is restored to full operation. Therefore, MTTR includes the times for failure detection, craft dispatch (if any), fault diagnosis, fault isolation, the actual repair, and any software resynchronization time needed to restore the entire service. For a simplex card (that is, without redundancy), the following relation is true:

Availability = MTBF , (MTBF + MTTR) for a simplex module

However, redundancy can reduce downtime by orders of magnitude (or adding one or more 9s to availability) while keeping the MTBF and MTTR the same. How effectively redundancy can improve availability is highly dependent on the switchover coverage.

NSF

Non-Stop Forwarding. The ability of a router to continue to forward traffic toward a router which may be recovering from a transient failure. Also, the ability of a router recovering from a transient failure in the control plane to continue correctly forwarding traffic sent to it by a peer.

Primary RP

Deprecated. Replaced by “Active.”

POS

Packet Over SONET.

PPP

Point to Point Protocol; RFC 1661.

PPPoA

PPP over ATM.

PPPoE

PPP over Ethernet.

PVC

Permanent Virtual Circuit.

RF

Redundancy Facility: A structured, functional interface, used to notify its clients of Active and Standby state progressions and events.

RP

Route Processor

RPR

Route Processor Redundancy - the feature formerly known as EHSA.

RPR+

Route Processor Redundancy Plus - An enhancement to RPR / EHSA in which the standby processor is fully initialized. An RPR+ switchover does not involve linecard reset nor linecard software reload.

RSP

Route Switch Processor. RP card on the c7500.

Secondary RP

Deprecated. Replaced by “Standby”.

Simplex

A system state in which only a single RP of a redundant pair is operational. Normally, this RP will operate as the Active RP.

Split

See maintenance mode.

Standby RP

The Standby RP. For SSO this is an RP that has been fully initialized and is ready to assume control from the Active RP should a manual or fault-induced switchover occur. The Standby RP is fully participating in stateful synchronization.

SVC

Switched Virtual Circuit.

Switchover

An event in which system control and routing protocol execution is transferred from the Active processor to the Standby processor. Switchover may be a manual operation (that is, CLI-invoked) or software/hardware initiated operation (hardware or software fault induced). Switchover may include transfer of the packet forwarding function as well in systems which combine system control and packet forwarding in an indivisible unit.

24.1.2 What is HA?

High Availability (HA) is defined as a percentage of time that the system is available for use, with a working definition of *available* as:

- The system is manageable
- The system has discovered peer routers and has formed routing adjacencies (insofar as it is configured to do so; static routing could be used) with its peers

- The system has negotiated and established connections with peer routers (again, as it is configured to do so)
- The system is forwarding subscriber traffic

The fraction of time a system is available is a function of the frequency of failure (MTBF) *and* a function of the time required to effect a repair for a given failure (MTTR). Specifically, for a nonredundant system, the following holds:

$$A = \frac{MTBF}{MTBF + MTTR}$$

Thus, availability may be improved by increasing MTBF, by decreasing MTTR, or both. Refer to ENG-36854 for background and partial derivation of this equation.

Use of redundant components can increase availability without change in MTBF or MTTR. What difference the use of redundancy has on availability depends on the time required to detect a failure, the time required to substitute the standby component for the failed component, and the likelihood that the substitution will be successful. When redundancy is used as a means to provide higher availability, methods which increase the probability of a successful switchover between the redundant components will directly improve availability.

A common expression of equipment availability is “number of nines,” for example:

- 99.99% available = “Four nines”
- 99.999% available = “Five nines”
- 99.9999% available = “Six nines”

Availability of five nines corresponds to an unplanned network or element downtime of roughly five minutes per year.

24.1.3 Assumptions & Requirements

The approach reflects several assumptions about the host router platforms and the surrounding network environment:

- 1 General system control is centralized in a single route processor.

The RP runs routing protocols, presents the system management interface, and generally manages the system.

- 2 Two instances of the RP exist, with one providing backup for the other.

While the RPs do not load share in the target platforms and load sharing is not a goal of the program, load sharing is not strictly precluded. Initial versions of the HA infrastructure software are unlikely to easily support load sharing.

- 3 Both RPs run IOS. Both IOS images are fully initialized.

In the future, as part of a software migration plan, it may be possible to have one processor run IOS and another a different operating system. We assume IOS runs on both RPs for simplicity.¹

- 4 Only one RP is active at a time. The active RP presents the image of a single router to an external operator, network management station, and routing protocol peer.

The precise definition of “active” will vary among platforms, but generally the active RP controls the system, exchanges routing information with peers, and implements the system management interface. As a result the system appears as a single router to adjacent systems and to external network management facilities.

1. For this to work the card state models used by the two operating systems must be consistent.

5 A data path connects the RPs.

This data path may be used to transmit events and synchronize state information between the RPs. Data flow is typically from the active RP to the standby RP. The data path is assumed to offer reasonably low-latency, high-bandwidth transport. It is not required to be reliable; whether it is so is platform-dependent.

While the data path may or may not be dedicated to transport of control information, a dedicated path is preferred. A 10 Mbps Ethernet may be sufficient, a 100 Mbps or faster connection would be better.

Each platform in the initial target platform set (7500, 10000, GSR) is thought to meet this requirement.

6 The versions of the IOS images running on the RPs are compatible with respect to state and event synchronization for each supported feature and protocol. Alternatively, a capability exchange mechanism may be used to allow the RPs to advertise the list of features each supports to the other.

Note It may be possible that the IOS versions are sufficiently different so that switchover will be service-affecting for features which supported non-service-affecting switchover previously. In this case, the CLI provides the operator with a way to determine what features support non-service-affecting switchover and which do not.

7 Peer routers support non-stop forwarding (as defined in the Non-Stop Forwarding Program Plan ENG-63195).

When the peer routers run Cisco IOS this assumption may hold true. However, the feature would need to be publicly documented for non-IOS peers to support the feature. External documentation and/or standardization of NSF features is an important program issue but is not considered here.

8 Distributed Cisco Express Forwarding (dCEF) is enabled on the platforms. dCEF is required for Non-Stop Forwarding.

24.2 Stateful Switchover (SSO)

IOS High Availability Stateful Switchover (SSO) combines Route Processor redundancy with IOS software enhancements to provide containment of software and hardware faults, to allow non-disruptive (or minimally-disruptive) switchover from the Active RP to a redundant “hot” Standby RP to provide an increase in network availability. The requirements for IOS SSO are described in 24.1.3 “Assumptions & Requirements.” In an SSO system, “HA aware” protocols and features may synchronize events and state information from the Active to the Standby RP so that the Standby RP remains in a state of “hot standby” with respect to the particular feature or protocol. Most often, a protocol or feature which implements such state synchronization will utilize the services of the IOS Redundancy Facility (RF), Checkpointing Facility (CF) and other IOS facilities designed to support SSO and NSF. Stateful protocols (Point to Point Protocol (PPP), Frame Relay (FR), Asynchronous Transfer Mode (ATM), Open Shortest Path First Protocol (OSPF), Border Gateway Protocol (BGP), Intermediate System-to-Intermediate System Interior Gateway Routing Protocol (IS-IS), Cisco Express Forwarding (CEF) will maintain state using either checkpointing or by recreating the necessary state after switchover by retrieving the state information from its peers. The approach used by a component depends on its architecture and other protocol considerations.

24.2.1 Platform Hardware Model

The HA SSO architectural system model is based on several fundamental architectural assumptions. This section describes those that are required of the platform hardware. The three main target platform architectures are described in detail in section 24.7 “Target Platform Architectures.”

24.2.1.1 Architectural Assumptions

The HA SSO architectural system model is based on several fundamental architectural assumptions. First is that there exists, typically within the “box”, redundant route processors (RPs) that run IOS. One of these RPs assumes control of the network interfaces and provides service (this one is known as the “Active” RP) while the other RP operates as a fully initialized mirrored copy waiting to take control should a fault occur or a manual switchover request be made on the Active. This “RP-in-waiting” is known as the “Standby” RP. Stateful IOS services and protocols running on the Active checkpoint state data to the Standby ensuring that it is always current and capable of taking over where the Active left off when a switchover takes place. In some architectures, the ESR for example, a Forwarding Processor (FP) is packaged with the RP so that they fail as a unit. In such architectures, the FP packaged with the Standby RP must be kept synchronized with the FP packaged with the Active RP.

A second assumption is that the physical interfaces, from the point of view of an observer outside the box, do not lose their signal during a switchover. This is critical as the purpose of SSO and NSF is to maintain traffic forwarding capabilities and protocol peer connections through a switchover event; peer protocols will not “see” an interface reset prompting them to declare their peer down and drop all session traffic through the failing router. On current systems this is achieved using line cards which are separated from the failing RP package and are connected via a shared bus or fabric interconnect to both RPs. The line cards are claimed by the Standby RP as part of the process of becoming the new Active thus preserving the interfaces intact. While this is not the only way to achieve the goal, it illustrates the concept through a particular implementation.

A third assumption is that there is a direct data path between the RPs that has sufficient bandwidth to support the checkpointing and other IPC traffic necessary for state maintenance in an SSO system. This interconnect might be a dedicated Ethernet interface, some fabric interconnect or an internal bus, but it should be sized to support peak bandwidth requirements experienced during initialization or periods of high connection rate when a great deal of state information is being synchronized. Ideally this interconnect should provide at least two paths since it is a critical single point of failure in the system. This communication path is best implemented as an out-of-band data path so that it is available at all times and so that it does not impact the routed traffic being serviced by the router.

A failure status bus or register which allows direct detection of RP faults or system reset events is desirable. This ensures that the need for a switchover can be detected quickly and accurately by the Standby RP reducing the period in which forwarding and protocol services are unavailable.

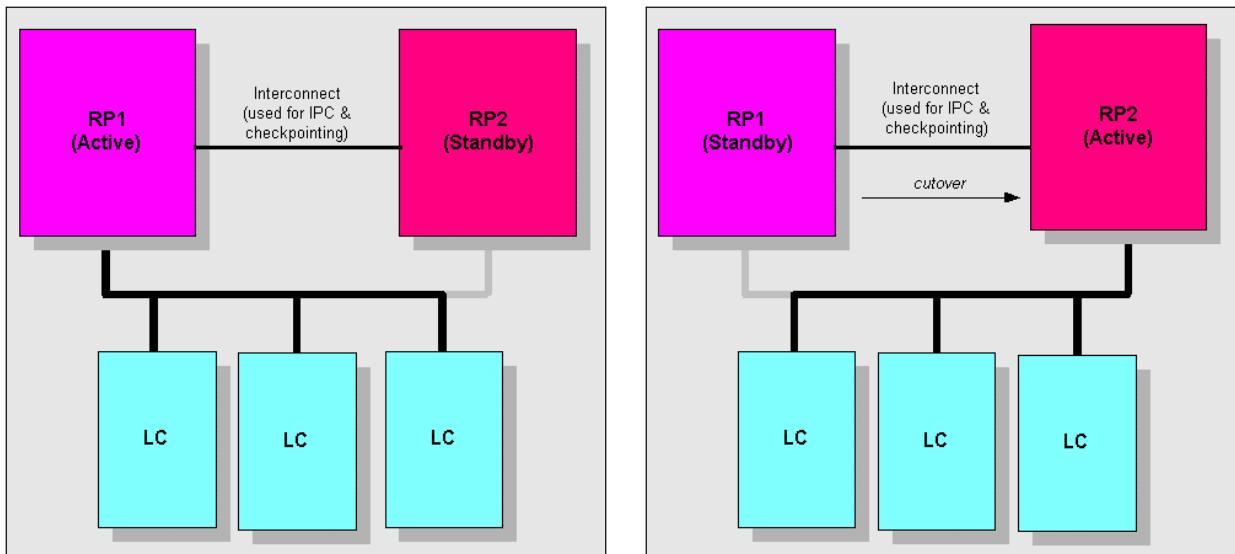
Hot swap support (OIR) for RPs as well as Line Cards is also desirable.

24.2.1.2 SSO System Platform Architecture

As has been described, an SSO HA system consists of redundant RPs interconnected via some media with sufficient bandwidth that it can be used for IPC and checkpointing messages between the Active and Standby RP. This may or may not be a dedicated media. Each RP is also connected to the Line Cards through some system bus or other interconnect. The Line Cards must be sharable (i.e., accessible to each RP) over this interconnect although their access may be serially restricted; only the Active may own and operate the Line Cards providing service at any point in time. When a service affecting fault occurs on the Active (or a manual switchover is requested) the Standby RP begins the process of assuming control of the Line Cards and transitioning the Standby applications

to Active so that the Standby can begin providing service as the new Active. Figure 24-1 demonstrates a system with the specified attributes before and after a switchover. Here RP2 becomes the new Active after the switchover and assume control of the Line Cards.

Figure 24-1 Basic HA SSO System Platform Architecture



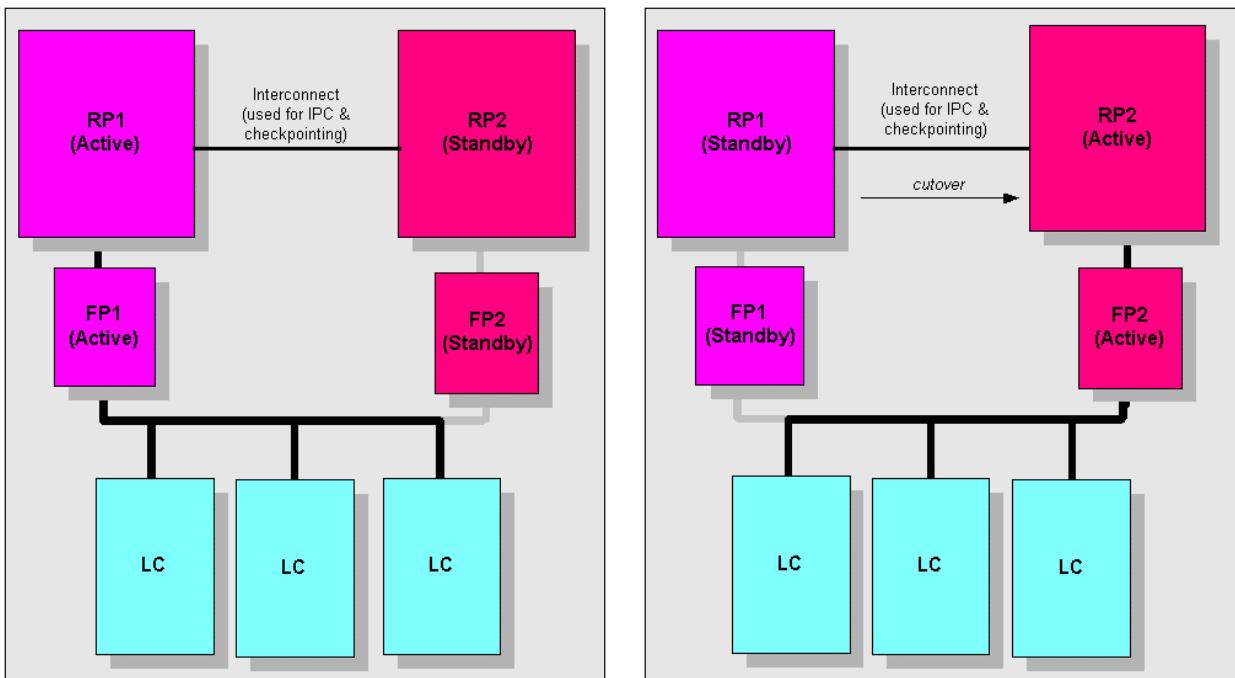
The Line Cards will continue forwarding in most implementations almost immediately after the switchover occurs. This is enabled by the Forwarding Information Base (FIB) data being distributed to the Line Cards (the case of an independent Forwarding Processor is discussed further below) which can then continue to switch traffic either on the same card or between cards assuming the cards have continued access to a data path between them during a switchover (this is not true for the 7500, for example, impacting its forwarding recovery time).

RP1, the failed RP, will re-boot itself and re-join the configuration as the new Standby RP. It will be brought up to date and will continue to act as the Standby until another switchover event occurs.

Some platform designs (e.g., the ESR) incorporate a Forwarding Processor as part of the platform architecture. In these designs the forwarding database is kept in the FP rather than on the Line Cards and the FP assumes control of the Line Cards. If the FP is packaged with the RP, as it is on the ESR, it fails with the RP and a new RP-FP pair assume control as the new Active. The FP associated with the Standby RP is kept current with the FIB (the mechanism is managed by CEF and is described further in the software section) and, once it has control of the Line Cards after a switchover, it can begin forwarding immediately. On the ESR this is projected to take less than three seconds. On the GSR, it is projected to happen almost immediately since all resources are available. The 7500 will take an extended period to begin forwarding again because of platform hardware requirements.

Although current platform designs that are part of the program do not package the RP and FP separately, there are existing designs that do (e.g., AS5850). It is far simpler to support failover in a system which has “simple” failure modes; increasing the number of possible configurations that must be supported during and after a failure makes HA implementations more complex and potentially less reliable. That said, it is possible for a platform to support many different failure and recovery configurations within the scope of HA. Figure 24-2 shows a system with an FP which fails with the RP. Forwarding continues after the switchover using FP2.

Figure 24-2 Basic HA SSO System Platform Architecture with an FP



24.2.1.3 RP Switchover Behavior

Features and protocols which support SSO are referred to as “HA-aware.” For these features and protocols, the state of negotiated circuits, routing protocol adjacencies, and traffic forwarding is maintained through a switchover from the Active RP to the Standby RP.

Features and protocols which do not support SSO (i.e., have not been modified for SSO) are referred to as “HA-unaware.” The goal for these features and protocols is that will continue to behave in the way that customers expect. Typically, they will mimic what happens today following a router restart or link flap.

24.2.1.4 Line Card Behavior for SSO

The line cards are expected to exhibit specific behavior in order to support SSO. This behavior includes the following:

- Line cards must not be reset as part of switchover. Those that are will not participate in SSO.
- Line cards must not be reconfigured as part of switchover, unless the standby RP configuration differs from the former active RP configuration, in which case only the differences in the two configurations should be applied. In some architectures, some line card configuration information may need to be updated because of the change in ownership after the switchover. These changes in configuration cannot affect existing sessions and forwarding. If they do, the line card does not participate in SSO, at least for the affected interfaces. For HSU, where a line card is supported only in the upgraded IOS image, the line card will not participate in SSO.
- Subscriber sessions may not be lost as part of an RP switchover.
- Line cards must not generate physical layer alarms as a result of an RP switchover.

- Line cards may periodically generate status events and transmit these to the Active RP. Status information includes line up/down status and alarm status. This supports bulk sync after Standby RP initialization and supports simple state reconciliation / verification after a switchover.
- For those architectures which require it, line cards must support online Standby test coverage (e.g., loopback).
- Line cards must clear statistics reported to IOS to zero upon a switchover. The impetus to clear statistics could be a message sent from the IOS driver to the line card, or could be the line card noticing that a switchover has taken place. This decision is platform specific.
- Line cards should accept and process received configuration messages in a first-come-first-applied order.

24.2.1.5 RP Switchover Conditions

The conditions under which a switchover may occur are described below. Some of these will not be implemented in the first release of SSO.

24.2.1.5.1 A Fault on the Active RP – Automatic Switchover

When a hardware or software fault occurs on the Active RP, it will cause an immediate switchover to take place. The implementation is platform specific.

24.2.1.5.2 Active RP Declared “Dead” – Automatic Switchover

A failed Active RP can be recognized by the Standby RP when the Active RP no longer responds to polls or no longer produces a heartbeat. The method of recognizing this condition will typically be platform specific.

24.2.1.5.3 CLI Invoked – Manual Switchover

The operator can force the switchover from the active RP to the redundant RP via CLI command. This manual procedure will allow for a “graceful,” or “controlled,” shutdown of the Active and switchover to the Standby. This “graceful” shutdown is intended to allow critical cleanup, such as flushing queued checkpoint messages for example, to occur. This will help make manual switchover a more deterministic process than an automatic switchover which might, for example, lose a checkpoint message in progress. It is not intended as a general last gasp notification mechanism for all applications. This should not be confused with the “Graceful Shutdown” procedure for routing protocols in core routers – they are separate mechanisms and this one is not intended to apply to the routing protocols.

24.2.1.5.4 CLI Invoked – Trial Switchover

This is intended to mitigate the failure of the Standby when a manual switchover is requested on the Active. It cannot be used when an automatic switchover occurs. It is intended to be used when a switchover is initiated to the Standby and it is determined to be faulty. The system should automatically switch back to the original Active unit without causing any additional outage.

This feature is perceived to be useful when:

- Some classes of errors on the Standby cannot be detected until live traffic is switched to it.
- To ensure that the switchover does not cause an outage if the operator wishes to perform a routine maintenance switchover.

Implementation of this feature is problematic because it requires that a new state be defined on the Active that is neither Active nor Standby and is not stateful. While the Standby is in the process of establishing itself as the new Active, stateful conversations are typically being re-established, but this assumes an Active-Standby relationship. Trial Switchover requires something new and different. A switch back to the former Active would not satisfy the switchover times required to maintain protocol peer sessions and the question of how long to wait is a difficult one.

Trial Switchover will not be supported in the first release. The approach used will be to reboot the former Active to become the new Standby ready to assume control should the new Active fail.

24.2.1.5.5 Criteria-based – Automatic Switchover

One or more hardware or software errors (not faults) may cause platform dependent or platform independent software to decide that a switchover is required. An example of a hardware error that might cause a switchover is some number of spurious recoverable memory parity errors crossing a predefined threshold. An example of a software condition that might induce a switchover a pre-defined time of day is reached. This will not be supported in the first release.

24.3 SSO Software Features

The HA SSO software architectural system model is based on some fundamental architectural assumptions and relationships. This section describes them in some detail.

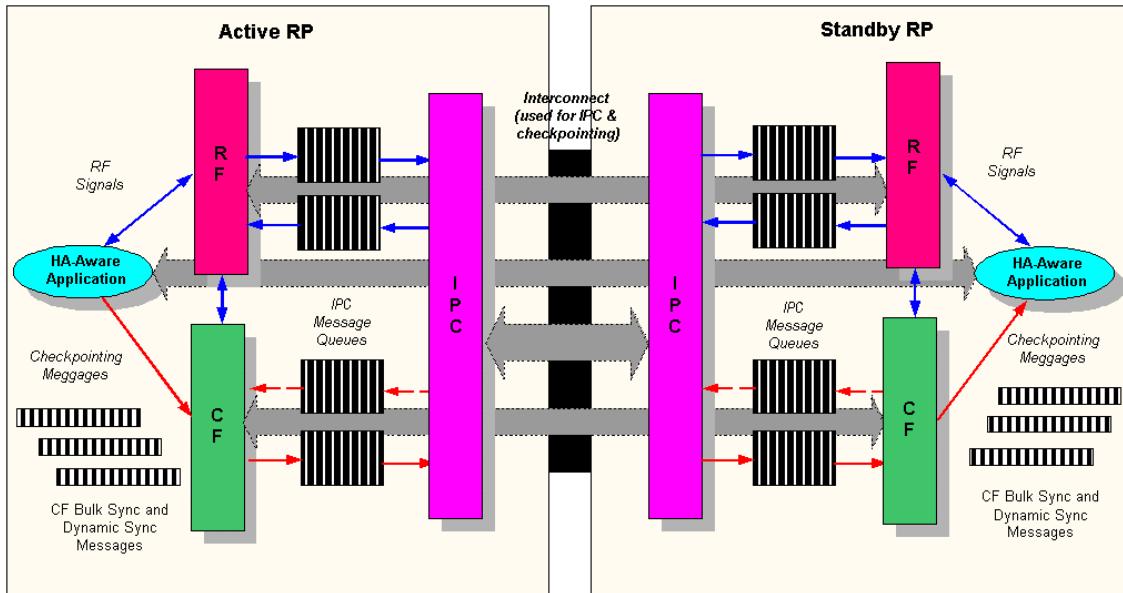
24.3.1 SSO Software Architectural Model

The software architecture for an SSO system is based on the two assumptions. First, the configuration must be synchronized between the Active and Standby RPs. The startup config, running config and dynamic config are all copied from the Active to Standby RP during execution by Config Sync. Dynamic configuration is the result of commands entered at the Active. These are passed to the Standby and executed there in order to create the same state there that exists on the Active.

The second fundamental assumption is that HA-aware features and protocols do not execute as they do on the Active. Instead they maintain the same state that is developed by their Active peer counterpart by waiting to receive synchronization messages containing the critical state from the Active process and then applying them at the Standby. This synchronization is referred to as checkpointing. The peer understands that it is executing on the Standby and that it is responsible for state maintenance until a switchover occurs. All HA-aware peers are also clients of the Redundancy Facility (RF), which operates on each RP and guides the clients through the transitions of that processor to its role as the Active or Standby RP.

RF and the Checkpointing Facility (CF) use IPC to communicate between their counterparts on the redundant RPs. This infrastructure provide the basis for the HA-aware clients to implement an SSO system. Figure 24-3 shows this relationship.

Figure 24-3 Basic HA SSO System Software Architecture



24.3.2 The Driver-Client Architectural Model

The model by which line card drivers track the status of the line card interfaces on the Active so that the Standby is able to recover the interfaces when a switchover occurs and make them available to services and protocols is based on a stateful interface model for hardware IDBs. Depending on the platform and/or the line card, either the platform dependent drivers or the platform independent media layer copy the interface state from the IDB on the Active to the IDB on the Standby. What happens on the Standby when a switchover occurs is described by two alternative models, dubbed “model 1” and “model 3.” They offer different HA-unaware client behavior on the Standby along with different risks as well as scaling and switchover performance characteristics. Figure 24-4 illustrates the general method of operation discussed in the following sections.

24.3.2.1 Model 1

With model 1, the interface state is copied to a “shadow” state variable in the IDB. The actual IDB state is marked “down”. This allows HA-unaware applications to remain unchanged and working with a well understood model – a system without any interfaces available (when they become available after switchover, they look like they’ve been OIRD in). It is assumed that any protocol which works with RPR+ will work with SSO using this model since it is very similar.

HA-aware protocols typically do not need to access the state while in Standby mode (although they are free to interrogate or be informed of changes to the shadow state if they require the information). They do not attempt to access interfaces; they are processing state synchronization messages to keep their session information current while the IDB information is being kept current by the driver or media layer.

At switchover, the shadow state is immediately copied to the “real” IDB state and the HA-aware protocols and services check the state to ensure that the interfaces are as expected by their current sessions. They can then use the interface to exchange keepalives with their network peers so that network connectivity is maintained. Existing session traffic is being forwarded by the line cards

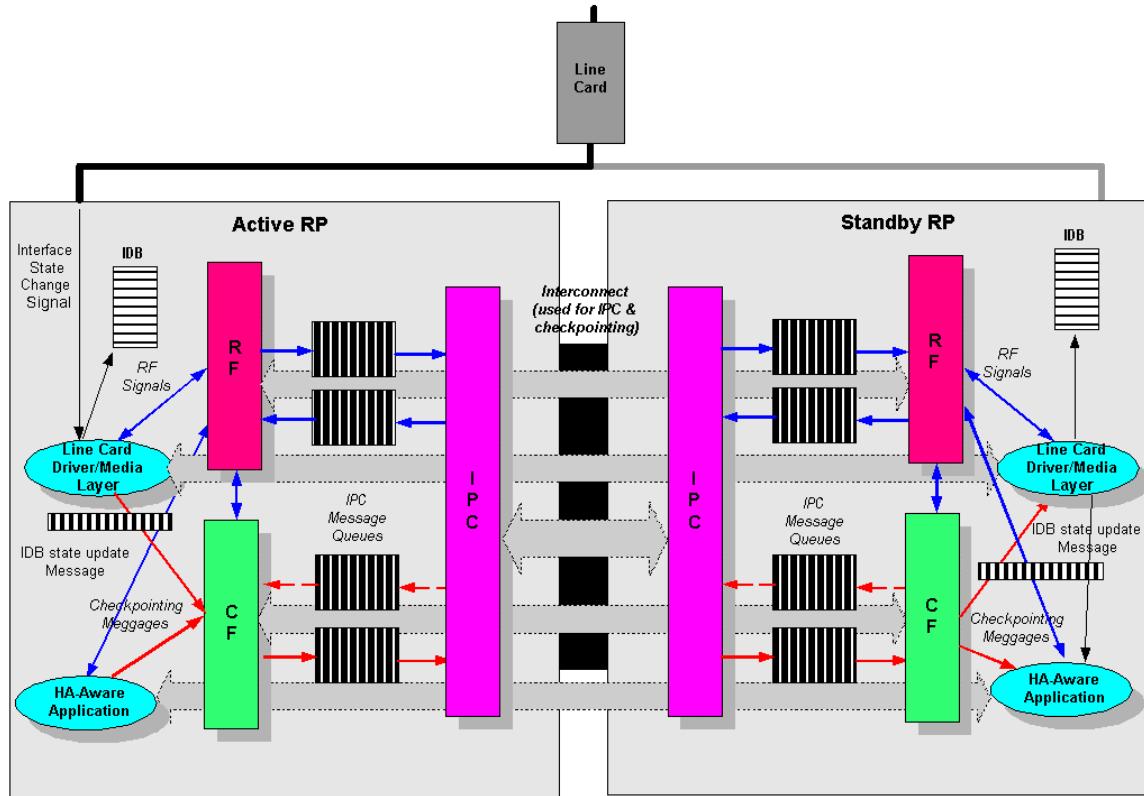
and/or FP. HA-unaware protocols and service are notified that the interfaces are available via the normal mechanism and begin contacting network peers as if the line cards had just been inserted on an Active system.

Model 1 allows the IDB shadow state to be maintained by one of two methods – either the Media Layer or the platform line card driver(s) maintain the IDB shadow state. These are selectable either on a per IDB or system-wide basis. The Media Layer will always be responsible for notifying the clients of a state change on switchover.

The issue with model 1 is that there is some per-IDB cost associated with the switchover to notify the non-HA aware clients that the interface is available. This should not affect the HA-aware clients much since it is delayed until after the HA-aware clients are informed their interfaces are “up.” Diagnostics can act as they do today. This satisfies the goals of SSO that the HA-unaware clients perform at least as well as they do in an RPR+ system and the HA-aware clients perform well enough to maintain connectivity. Threshold scalability and performance testing, against known platform goals, needs to be done to understand when this model will not sufficiently scale and/or perform to satisfy target platform goals. The system has to be instrumented and the results have to be analyzed to distinguish and fix non-fundamental performance/scaling problems from fundamental performance/scaling problems.

The prototype, using model 1, performed within the expected switchover times for the configurations that the testers had available and could drive; the session limits were not at the target thresholds for the platform but were significant (e.g., 600 sessions for PPP). The problems encountered (it was a prototype) were not fundamental. The model that the driver-client working group decided to go forward with is “model 1.” It is the lowest risk model and worked reasonably in the prototype implementation. Figure 24-4 illustrates the basic driver-client model:

Figure 24-4 Basic Driver-Client Model



24.3.2.2 Model 3

Model 3 is the model we will eventually evolve towards. In this model, the IDB state on the Standby represents the actual state of the interface on the Active. It is still synchronized but is copied directly to the IDB state rather than to a shadow state. It has the advantage that all of the HA-unaware applications would be completely initialized and actively “using” the interfaces at switchover. It has the disadvantages that it is unknown how these HA-unaware applications (at this point, the bulk of the protocols) will act in a 100% lossy network over a long period of time. Their recovery actions in the face of a constantly non-responsive network is unknown. Some may have to be “fixed” and/or made somewhat HA-aware. All of the generated packet traffic must be captured and quietly thrown away. The testing requirement here is beyond the regression testing required for model 1 and represents significant work. How to detect and provide for diagnostic packets as opposed to protocol packets without modifying the applications is problematic. Diagnostics which find the interface non-responsive may declare the Standby down. It was decided that this model represents more risk.

24.3.2.3 Model 1 vs. Model 3

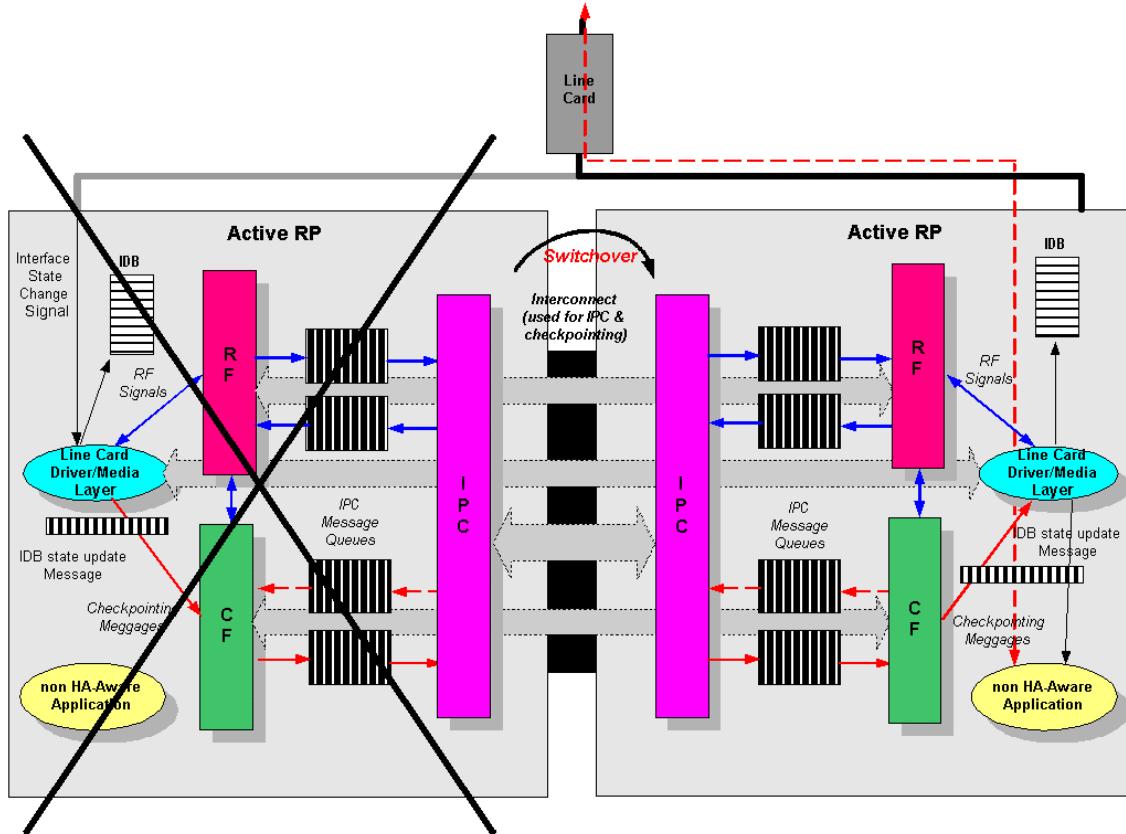
The bulk of the driver-client model implementation is really outside the model 1 vs. model 3 discussion. The execution model remains the same. A change from model 1 to model 3 will affect the HA-aware protocols little or not at all. The decision for model 1 vs. model 3 centers around when and if model 1 fails to scale to the numbers and size that the platforms require. At the point that the curves cross, we need to be ready with a model 3 implementation. The first step is collecting performance and scaling information on model 1 so we know when we have to start on model 3.

The driver-client working group felt that model 1 would scale sufficiently for at least the initial platform goals and should be used for EC estimates.

24.3.3 Non HA-Aware Protocols and Features

The objective of the architecture is to leave the non HA-aware protocols unmodified until they are made HA-aware. This allows for a low-risk and smooth transition based on business decisions. To achieve this, the non HA-aware protocols and features must perform at least as well as they do during an RPR+ switchover and their behavior must be similar to that on a non SSO system. Figure 24-5 shows a switchover for a non HA-aware application assuming model 1.

Figure 24-5 Non HA Aware Protocols and Features on Switchover



24.3.4 Software Upgrade

Software upgrade is an important consideration for HA. Planned upgrades account for a significant DPMs in customer configurations and any way to lower these DPMs is a big win.

There are currently two proposed methods to achieve planned software upgrades within an HA system. The first is called Fast Software Upgrade (FSU) and the second is called Hitless Software Upgrade (HSU). FSU works with systems that have at least RPR installed. HSU is based on an SSO system and will replace FSU for those systems that implement HSU.

24.3.4.1 Fast Software Upgrade (FSU)

FSU will work with HA systems including RPR and RPR+. It will continue to work with SSO systems. Fast Software upgrade provides a means to upgrade (or downgrade) the Standby image and allow it to initialize to the point where it is “warm.” The operator then has the choice to cause a manual failover to the new image on the Standby.

When the new image is booted, it will “handshake” with the Active during initialization and compare versions of the IOS image and operating mode. If the versions differ (which will be the case during an upgrade) or the operating mode is not understood, the system will force an operating mode equivalent to RPR.

24.3.4.2 Hitless Software Upgrade (HSU)

HSU is built on top of an SSO implementation. When implemented, it replaces FSU and allows two otherwise incompatible IOS images to operate in SSO mode. The goal of Hitless Software Upgrade is to enable upgrading or downgrading the code image on a router with no loss of the sessions on the router. Version control and chained transformations must be implemented on all RP to/from RP and RP to/from LC conversations to automatically handle any differences between the versions.

This feature will be delivered on some platforms after SSO. For more detailed information., see “HSU Investigation Notes.”

24.4 SSO Implementation Overview

24.4.1 SSO Platforms

The initial platforms targeted for SSO HA are the 10000 (ESR), 12000 (GSR) and 7500. The ESR has been used as the prototype development platform and will likely deliver the SSO functionality ahead of the other platforms.

24.4.2 SSO Software Features

There are a number of software components involved in the initial SSO delivery. The modifications and new functionality added to each of them is described in the sections below along with a pointer to the more detailed references.

24.4.2.1 Routing Protocols and Non-Stop Forwarding (NSF)

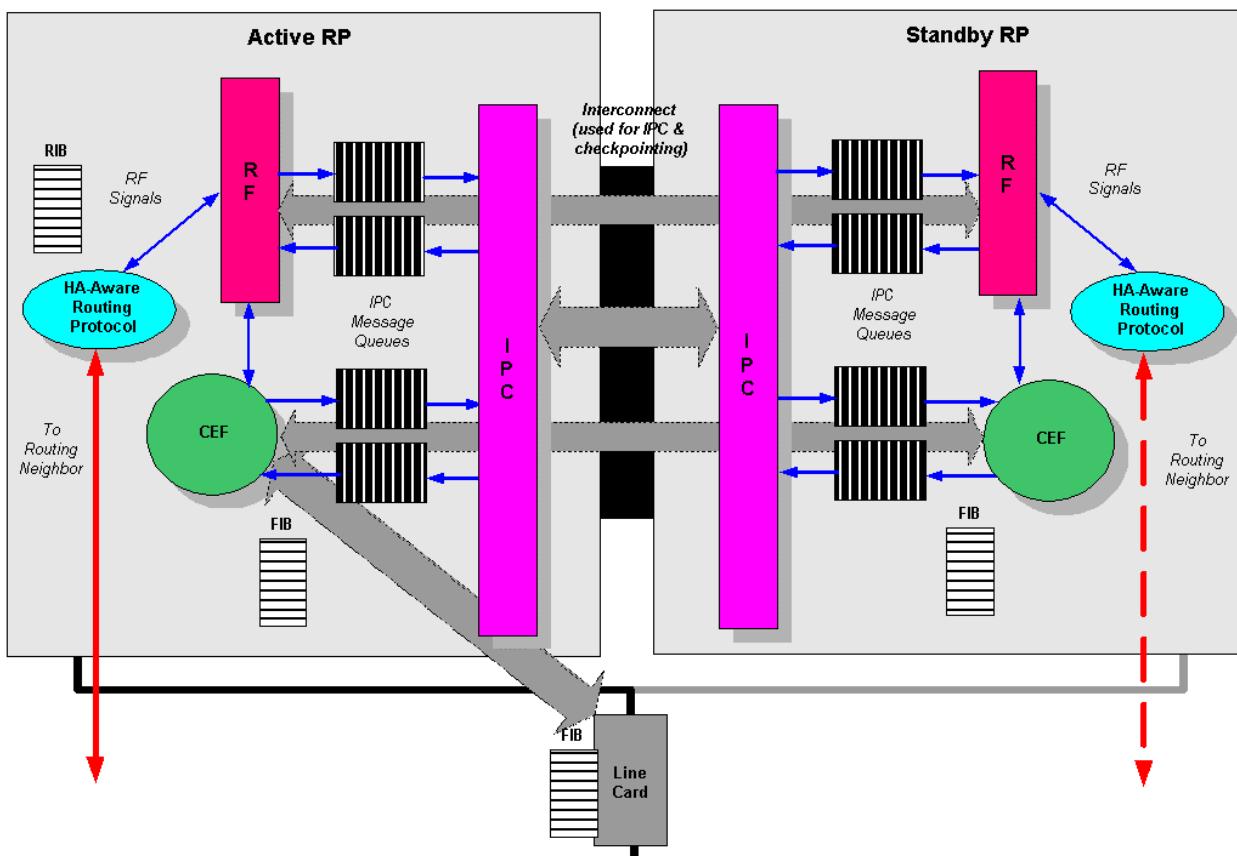
Non-Stop Forwarding (NSF) is an effort to minimize the amount of time a network is unavailable to its users. Usually when a router restarts, all the routing peers of that router detect that the router went down, and then came back up. This “down/up” transition results in a “routing flap.” The flap could spread across multiple routing domains.

Routing flaps caused by router restart create routing instabilities, which in turn can create transient forwarding black holes and/or transient forwarding loops. They also consume resources on the control plane of the routers affected by the flap. They are detrimental to the overall network performance. NSF allows suppression of such routing flaps thus reducing network instability.

NSF allows for the forwarding of data packets to continue along routes that are already known while the routing protocol information is being restored. NSF will be implemented for OSPF and BGP first, with ISIS and other protocols to follow. A redundant, HA-aware CEF is an integral part of the NSF implementation. The basic idea is to continue to forward data traffic through intelligent line

cards or redundant forwarding engines (FPs) while the Standby RP assumes control from the failed Active RP during a switchover. The ability of the Line Cards and FPs to remain up through a switchover and to be kept current with the Forwarding Information Base on the Active is key to NSF operation. Eventually the routing protocols on the new Active RP will converge and the stale forwarding information will be purged eliminating old forwarding information that may have temporarily caused block holes or routing loops. Figure 24-6 illustrates the relationship between Routing Protocol Redundant Pairs and CEF in an SSO system. The Routing protocols in the first release do not perform state updates between the Active and Standby copy. Instead, when the switchover occurs, the Routing Protocol uses existing or new protocol elements to recover its state from its peer(s).

Figure 24-6 HA Aware Routing Protocols



24.4.2.1.1 Cisco Express Forwarding (CEF)

Packet forwarding in a Cisco router is provided by Cisco Express Forwarding (CEF) and the Forwarding Information database (FIB) that it maintains. CEF NSF will provide the required synchronization of data between the Active and Standby RPs to allow packets to be forwarded during a switchover.

Initially CEF will use the same mechanism used by distributed CEF (dCEF) to keep Line Cards synchronized with the FIB on the RP. The Standby RP will be treated like another Line Card from the point of view of CEF forwarding information distribution. During normal operation, CEF on the Active RP will synchronize its current FIB and adjacency databases with the FIB and adjacency databases on the Standby RP. Upon switchover of the Active RP, the Standby RP will initially have FIB and adjacency databases that are mirror images of those that were current on the Active RP. For

platforms with intelligent Line Cards, the Line Cards will maintain the current forwarding information over a switchover; for platforms with Forwarding Engines, CEF will keep the Forwarding Engine on the Standby current with changes that are sent to it by CEF on the Active. In this way, the Line Cards or Forwarding Engines will be able to continue forwarding after a switchover as soon as the interfaces and a data path are available. As the routing protocols start to repopulate the RIB on a prefix-by-prefix basis, the updates in turn cause prefix-by-prefix updates to CEF which it uses to update the FIB and adjacency databases. Existing and new entries will receive the new version (“epoch”) number, indicating that they have been refreshed. The forwarding information is being updated on the line cards/forwarder during convergence. Some time later when the RIB signals that it has converged, a database walker will remove all FIB and adjacency entries that have version numbers older than the current switchover version number. The FIB now represents the newest routing protocol forwarding information.

The NSF routing protocols depend on the CEF functionality having been implemented. For more information, see “IOS High Availability: CEF NSF Software Unit Functional Spec. (ENG-77161).”

24.4.2.1.2 Border Gateway Protocol (BGP)

In the case of BGP peers, a routing flap causes the generation of BGP routing updates, BGP route selection, and may also cause flapping of the forwarding tables. Three new mechanisms have been defined for BGP that would help minimize the negative effects on routing caused by BGP restart. First an End-of-RIB marker is specified and can be used to convey routing convergence information. Second a new BGP capability, termed “Graceful Restart Capability,” is defined which would allow a BGP speaker to express its ability to preserve forwarding state during BGP restart. Finally, procedures are outlined for temporarily retaining routing information across a TCP transport reset. This allows suppression of routing flaps. At the completion of this procedure, the router and its peers are resynchronized, with little or no route flap having been propagated and most user traffic having been unaffected.

BGP is HA-aware but is not stateful. As an RF client, it knows that it is running on the Active or Standby RP and knows when a switchover occurs. It recovers and recreates its state from its peers. It depends on CEF-enabled continued forwarding while it rebuilds its routing information. For more information, see “BGP NSF for IOS Software Unit Functional Spec. (ENG-72947).”

24.4.2.1.3 Open Shortest Path First Protocol (OSPF)

For standard OSPF, there is no way for neighboring routers to resynchronize their topology databases (Link State Databases – LSDBs), without causing the adjacencies to be removed from the neighbor LSDB entries, (Link State Advertisements – LSAs). All routers that belong to the “same area” have the same LSDB. The proposed solution is to modify the OSPF protocol to support out-of-band (OOB) LSDB resynchronization. This prevents the adjacency flap. It also means that NSF/OSPF will support NSF on same network segment as the reloading router only if all peers in that network segment support NSF/OSPF. It should be noted that there is a competing proposal to support NSF within the IETF called “Hitless OSPF Restart.” This proposal is incompatible with the existing Cisco-originated proposal.

OSPF is HA-aware but is not stateful. As an RF client, it knows that it is running on the Active or Standby RP and knows when a switchover occurs. It recovers and recreates its state from its peers. It depends on CEF-enabled continued forwarding while it rebuilds its routing information. For more information, see “OSPF NSF on IOS Software Unit Functional Spec. (ENG-66064).”

24.4.2.1.4 Intermediate System-to-Intermediate System Interior Gateway Routing Protocol (IS-IS)

ISIS was originally going to implement a stateful client. However current thinking is to propose and implement protocol changes to achieve IS-IS stateful switchover goals. This work is not yet complete. The current IETF draft proposal describes a mechanism for a restarting router to signal that it is restarting to its neighbors, and allow them to reestablish their adjacencies without cycling through the down state, while still correctly initiating database synchronization. Additionally a mechanism for a restarting router to determine when it has achieved synchronization with its neighbors is described.

IS-IS will not be in the first release of NSF for SSO and so it is not discussed further here. For more information, see “IS-IS NSF on IOS Software Unit Functional Spec. (ENG-66448).”

24.4.2.1.5 Graceful Shutdown

“Graceful Shutdown” is intended to be used in the core where routers are fully meshed to accelerate routing reconvergence on peer routers after a router outage. Prior to shutting down for a planned outage, a router can “gracefully” remove itself from the routing topology by whatever means is available to that protocol (e.g., the overload bit for ISIS, the stub router capability presented by Cisco at the recent IETF for OSPF, etc.). The neighbors will then route traffic around the router that has “gracefully” shut down until they get an indication from the router that it is ready to be added back into the routing topology. Reconvergence is accelerated because the neighbor routers don’t need to wait for timeouts to expire before declaring the failed router dead before they adjust their routing databases. This should not be confused with a “graceful” shutdown of an RP via an operator requested CLI command which is intended only to make switchover more deterministic by flushing checkpoint messages for example.

24.4.2.2 Routed Protocols

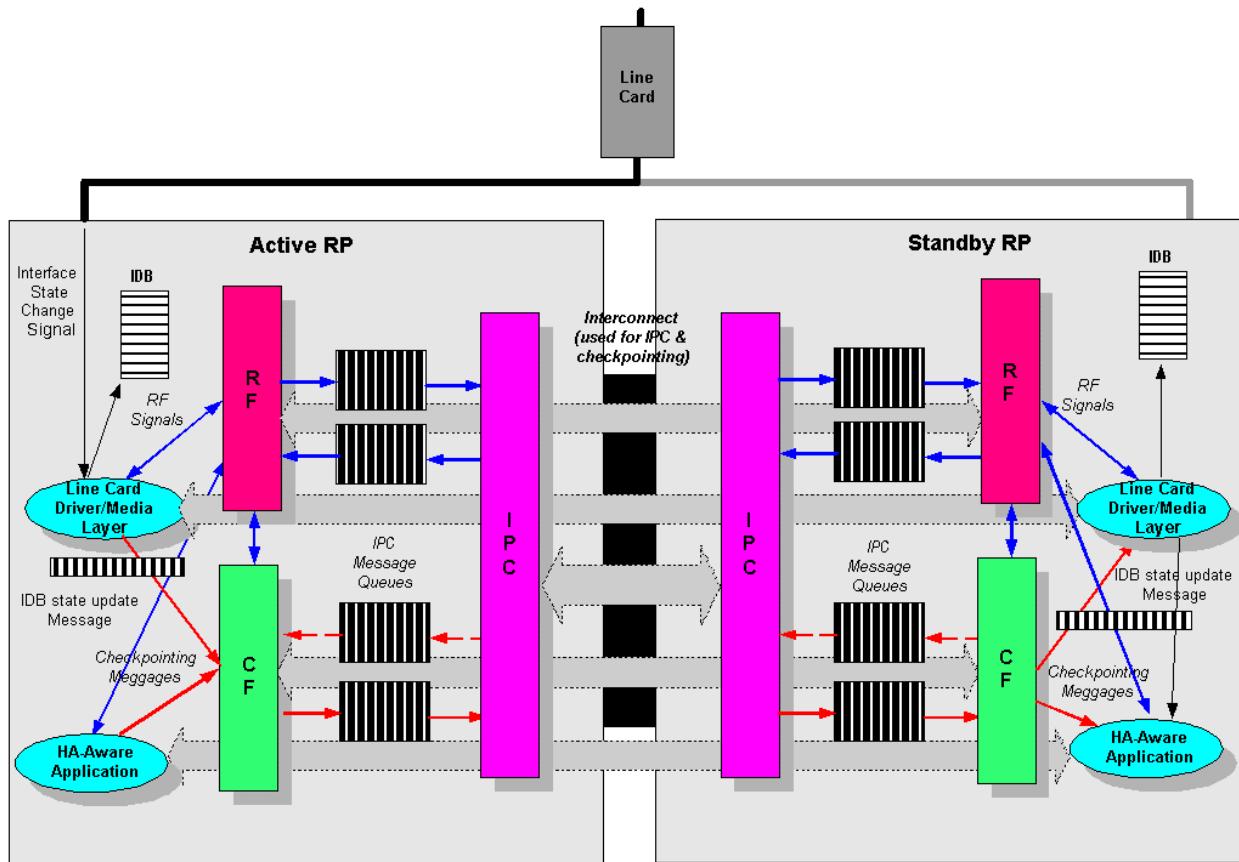
Several other IOS Connectivity protocols are being made HA-aware for the initial SSO release. These are discussed in the following sections.

24.4.2.2.1 Point to Point Protocol (PPP)

PPP for an SSO system provides a generic mechanism for the an initially limited subset of PPP capabilities with version independent state transfers. It is designed with extensibility for future PPP SSO capabilities. The scope of the design is limited to basic PPP functionality; dialer, AAA, IPPOOL, DHCP, MLP, L2X, PPTP, MPPE, PPPoE, PPPoA, LQM, link or header compression, bridging, asynchronous PPP, and XXCP (IPCP only is supported) are not supported. PPP in leased line and edge routing applications typically use statically defined IP addresses configured on the serial interface. The application does not require radius or IP pools for address negotiation. PPP transfers sufficient stateful information to the Standby RP to alleviate the need for LCP or NCP (in this case IPCP) to renegotiate on a given link. The design takes into account future requirements with MLP and forwarded PPP sessions and does not preclude future redundancy development in these areas. For more information, see “PPP High Availability Software Unit Functional Spec. (ENG-72990).”

Figure 24-7 illustrates HA-aware routed protocols on an SSO system:

Figure 24-7 HA-Aware Routed Protocols



24.4.2.2.2 Frame Relay (FR)

With SSO, Frame Relay synchronizes dynamic state. PVC state changes are not synchronized for either switched or terminated PVCs. PVCs are assumed to be up when a switchover occurs. Their true state will be determined when the first full status message is received from the switch. If switchover exceeds the LMI time out value at the switch because of lack of keepalives from the router, the switch might declare the LMI down and all PVCs with it. If this happens, the fallback is to use FR Fast Restart. FR Fast Restart improves the time taken for PVCs to come up and dynamic mappings to be established by assuming PVCs are up on switchover, accelerating the LMI cycle and accelerating Inverse ARP. For more information, see “Frame Relay High Availability Software Unit Functional Spec. (ENG-80485).”

24.4.2.2.3 Asynchronous Transfer Mode (ATM)

ATM will provide VC redundancy (PVC, PVP). It provides redundancy for the following software components:

- ATM software: VC Manager, OAM Manager, ATM InARP, etc.
- ATM Device Drivers: platform specific support for slot/port redundancy required.

The following are not supported in the first release: SVC/SVP, TVC, Point-to-Multipoint SVC, ILMI, signaling and SSCOP, ATM Connection Manager, PVC discovery, ATM Applications, PNNI, backward/version compatibility, not all MIBs, statistics and accounting, zero ATM cell loss.

For more information, see “IOS ATM HA Architecture and Software Unit Functional Spec. (ENG-104679).”

24.4.2.3 IOS Infrastructure Components

There are a number of IOS Infrastructure components that are either being modified or are entirely new in order to support client needs in an SSO system. These components are briefly described in this section.

24.4.2.3.1 Configuration Synchronization (Config Sync)

For SSO, it is required that both RPs are running the same configuration at any point in time so that the Standby RP is always ready to assume control for a failed Active RP. This is accomplished by synchronizing the configuration between the Active and Standby at several points.

The startup-config is synchronized when the configuration file is closed after it has been opened for a write operation. The running configuration is generated dynamically by traversing the entire parse tree and calling the command function to output the data associated with that command. It is copied to the Standby during the initial Bulk Sync. In order to avoid race conditions among RF clients, Bulk Config Sync must be completed before any other Bulk Sync RF clients begin to run on the Standby.

Ongoing configuration changes are synchronized by first executing the CLI command on the Active and then, if it was successful, sending it to the Standby to be executed there. There are two methods, “Parser Return Codes” and “Best Effort”, used to do this. This is known as line-by-line sync.

The Parser Return Codes approach is described below; this is implemented for all of the AT&T commands in the first release. Best Effort involves running a NVGEN for a given command before and after the action function is executed, saving the results into two different buffers, and if those two results differ, send the command to the Standby. Otherwise, don’t send the command to the Standby. This scheme is not foolproof, because some commands might not cause a difference in the NVGEN results when the command executes successfully. This scheme is used for any commands not yet converted.

For more information, see “Configuration File Synchronization for HA (ENG-78978).”

Parsing at System Startup

When the system starts up, it first parses the startup config for the following cases:

- In a non-redundant system, with no ROMMON override-config setting, the system first parses the startup config. After all, that is where the name “startup” config comes from—it is (or used to be) what gets parsed at startup.
- In a redundant system configured for HSA or EHSA (RPR), with no ROMMON override-config setting, the system first parses the startup config for both the Active and Standby, for most platforms. The exception is the 8540 (Cougar) with EHSA setting. In that case, the synced running config gets parsed on the Standby, after it switches over.
- In a redundant system configured for RPR+ or SSO/NSF, the system first parses the startup config for the Active but not for the Standby, subject to the ROMMON caveat. On the Standby, the synced running config gets parsed when it starts up, then running config gets generated by going through the whole parser tree and calling every EOL function with csb->nvgem true? This is done on a redundant system to generate the running config to be synced to the Standby, when the redundancy mode is RPR+ or SSO/NSF. This is done on an as-needed basis, every time a Standby comes online. Previously, this was done on an as-needed basis, every time a Standby came online.

24.4.2.3.2 Parser Return Codes

Without return codes, there is no programmatic way to determine if a given IOS command has executed successfully. This functionality is required for SSO so the Parser can tell whether a command executed on the Active has succeeded and therefore whether it should be passed to the Standby for execution. It also needs to monitor the success of the command on the Standby so that it knows whether the Standby remains configured exactly the same as the Active. If the command fails on the Active, then it is not passed to the Standby for execution since the configuration on the Active should not have changes. If the command fails on the Standby, then the Standby configuration is no longer in sync with the configuration of the Active and must be restarted. The Parser Return Codes project implements a way to retrofit return code support into the IOS CLI so that SSO can implement the necessary configuration functionality.

For more information, see Adding Parser Return Codes.

24.4.2.3.3 File System and Other Configuration Issues in an SSO System

There are a number of items in the file system which are related to the SSO mode of operation. These include: the Configuration register, the Boot image (in boot flash), Line Card images (bundled in with the IOS image), ROMMON variables, Startup configuration, Private configuration, and SNMP data (all in NVRAM). These items hold configuration information that may need to be the same on both RPs.

The decisions made regarding these various items are summarized here and in other sections of this document.

Configuration Register

The configuration registers need to be set the same on both RPs for the system to behave the same when either RP is rebooted. This will be documented but not programmatically enforced for SSO.

Boot Image

The default behavior for SSO mode is not to sync over the IOS image to the Standby RP. However, documentation will advise the user that both RPs must be running the same IOS image in order to enable the system to run operate in SSO mode. When the customer wants to upgrade the version of IOS that is running, the procedure described for FSU will be used and the system will not operate in SSO mode while the images differ. This will change for those platforms which implement HSU.

Line Card Images

Since the Line Card images are bundled with the IOS image, they are automatically the correct version for the IOS image that is running.

ROMMON Variables

The default behavior for SSO mode is to sync the Active RP's ROMMON variables to the Standby RP. This can be turned off so that it is possible for the Standby to behave differently after a boot in some situations. Documentation will advise the user to review ROMMON variables to verify that correct operation will result when a reboot occurs.

Startup Configuration

This is discussed in the section 24.4.2.3.1 "Configuration Synchronization (Config Sync)."

Private Configuration

The default behavior for SSO mode is to sync the Active RP's private configuration to the Standby RP.

The private configuration (directly from NVRAM) gets sync'ed to the Standby RP as part of the Bulk Sync, then the active Config Sync creates the **running-config** and sends it to the Standby RP via the rcsf file. At that point, the **private-config** gets parsed on the Standby RP as part of Bulk Sync. This behavior is fully documented in DDTS CSCsh56121.

SNMP information

The default behavior for SSO mode is not to sync over the Active RP's SNMP information to the Standby RP.

OS-logging

The default behavior for SSO mode is not to sync over the OS-log file to the Standby RP's file system for any reason.

System Memory

The default behavior for SSO mode is not to sync over the System Memory to the Standby RP.

Remote File Systems

The default behavior for SSO mode is not to sync over the information concerning the remote file systems to the Standby RP. This is typically FTP, TFTP or RCP URLs to be used for boot, etc.

24.4.2.4 Redundancy Facility (RF)

The Redundancy Framework (RF) is a portable framework that provides synchronization and switchover coordination between redundant processors running IOS. The RF infrastructure provides a series of client services as well as functions for control and monitoring of system redundancy. RF provides notification of transition events which occur on both processors to its clients. RF is key to the operation of a redundant system.

RF clients may maintain state synchronization between their Active and Standby instances. If they choose to maintain state, clients will use the Checkpointing Facility (CF) to do so. The format of client messages and the details of the synchronization protocol are client-specific. Initial synchronization is expected to be done as a result of an RF notification. Dynamic state changes, once the systems have reached the stable state (i.e., Active-Fast and Standby-Hot for the Active and Standby RPs respectively), will be synchronized at times decided by the client application.

A client may register with RF for notification of system state progression and changes in redundancy control configuration. These notifications are provided by RF in the form of synchronous and asynchronous callbacks. Clients are expected to use events to control the state of the applications as appropriate to their role at the time of the event.

For more information, see section 24.5 "Redundancy Facility."

24.4.2.5 Checkpointing Facility (CF)

In order to allow a Standby component to take up where its Active “partner” left off after a switchover, the Standby copy of the component must somehow have the same state data as its Active partner. The Standby copy can achieve this in one of two ways: it can either transfer state to the Standby as it changes on the Active or the Standby copy can recreate the state that the Active had (by retrieving it from some other source – a Line Card or peer partner on another machine for example). The Checkpointing Facility (CF) enables the first approach.

CF facilities provide the mechanisms to help synchronize client-specific state data, in a consistent, repeatable, and well-ordered manner. It is intended to be used wherever client state data needs to be synchronized; this includes RF clients when they receive RF events that indicate state should be exchanged as well as whenever the client state data changes during steady state execution once full Active and Standby states have been reached on the redundant processors. It is expected that all CF clients will also be RF clients although that is not strictly enforced or required.

CF provides a session-oriented API to its clients. It is intended to efficiently deliver state data changes from Active clients to their Standby counterpart. It is not intended as a general purpose two-way communications interface. IPC provides that functionality directly. CF will use IPC as the transport but will provide a simplified interface that is oriented to quickly delivering checkpointing messages from the Active to the Standby with a minimum of resource usage and little dedicated resource consumption. Its intent is to use as few resources as possible on the Active while delivering to the Standby as quickly as possible in a reliable fashion. Checkpointing will implement several optimizations in order to achieve this. The messages are considered “safe” once they have been delivered to the Standby system. The objective on the Standby is to deliver these messages to the CF clients as quickly as possible so that they are kept up-to-date thus minimizing the work to do on a switchover event. It should be noted that the semantics of the individual checkpoint messages are entirely up to the clients. The message data is treated as an opaque “bag of ordered bits” by CF. These messages may in fact be either state data updates or events that will cause the Standby client to create the correct state there.

For more information, see “IOS Classic HA Checkpointing Software Functional Spec. (ENG-78303).”

24.4.2.6 IPC

There are three pieces to the IPC enhancement plan:

24.4.2.6.1 Protocol Versioning

The current protocol definition for IPC is very rigid and cannot be modified easily to support new features required for the High Availability (HA) Initiative without breaking backwards compatibility. In order to provide new functionality and services in IPC while maintaining backward compatibility, a protocol versioning scheme will be implemented and enforced. The range of revisions for which interoperability with previous protocol definitions will be maintained will be determined at each new protocol revision. Revisions of the IPC protocol should occur infrequently. For more information, see “Cisco IOS Inter-Process Communication (IPC) for HA - IPC Protocol Versioning, SDS. (ENG-90337).”

24.4.2.6.2 Windowing

In order to improve the throughput of IPC reliable communication a sliding window algorithm with selective repeat error recovery is used in place of the current stop-and-wait algorithm for reliable IPC transmissions. For more information, see “IPC Sliding Window and Fragmentation Reassemble Support SDS (ENG-110718).”

24.4.2.6.3 Fragmentation and Reassembly (FAR)

Fragmentation and Reassembly provides support for IPC messages up to 4GB (232) irrespective of the seat's MTU size. Fragmentation and Reassembly support is provided at the IPC level and is transparent to IPC applications. FAR works in cooperation with Windowing. For more information, see "IPC Sliding Window and Fragmentation Reassemble Support SDS (ENG-110718)."

24.4.2.7 Media Layer (a.k.a. network.c)

Two mechanisms have been developed to synchronize interface state information from the Active RP to the Standby RP:

- 1 Platform-dependent device driver synchronization of line card events
- 2 Synchronization of interface state information by platform-independent code in the platform independent media layer code (a.k.a. network.c).

The second is described here and is used by the 7500 platform implementation.

The media layer modifications have been made for SSO in order to keep the IDB state from the Active synchronized with the IDB state on the Standby and to inform clients of the availability of the interface after a switchover. This will help to minimize the time required to restore service after a switchover.

The media layer is always involved in notification of clients but may optionally be selected on either an IDB or system-wide basis to manage IDB state synchronization. The other alternative is that the platform-specific drivers perform the synchronization. This alternative is discussed in section 24.4.3.1 "Line Card Drivers."

During the initialization of the Standby RP, IDBs are created, initialized and queued on the active IDB queue. The IDB state is marked "down" (i.e., either `IDBS_DOWN` or `IDBS_ADMINDOWN`). This presents a "well-known" model to the HA-unaware protocols on the Standby RP and allows them to act as they do on a system when interfaces are not available. For the purpose of interface state synchronization, a "shadow" IDB state is added. This reflects the state of the IDB on the Active RP. The shadow state can be used by the HA-aware protocols on the Standby RP as required. As part of the switchover, the "real" IDB state is updated with the IDB shadow state so that it now reflects the actual state of the IDB on the Active at switchover time. HA-unaware clients are informed of the state change after a switchover via the usual `net_cstate()` registry mechanism. HA-aware clients may also be informed of shadow state changes, if they choose, via the new `standby_hwidb_state_change()` registry mechanism.

24.4.2.8 Coredump

Coredump, after a system fault which causes a switchover to the Standby in an HA system, has been designed to use IPC and IFS/RFS to provide a generic implementation to support all HA platforms with a minimum of platform dependent code. The line card crash dump is not addressed in this design. The coredump file can be copied to the network, to the file system of the Standby RP or to the file system of the Active RP (that just failed and caused the switchover).

In an SSO system, the correct configuration options must be chosen for coredump so that:

- 1 It does not impact the Standby from taking over as the new Active by flooding it with messages while it is trying to restore service
- 2 It does not significantly impact the time it takes the failed Active to re-enter the configuration as the new Standby by rebooting.

The default behavior for SSO mode is not to sync over the crash dumps to the Standby RP or to the network. Taking the coredump to a local device on the failed system is the preferred model for an SSO system given that the platform has sufficient storage space and the device is sufficiently fast. In any case the implementation of coredump and other diagnostic facilities should not significantly impact or impair availability for SSO.

For more information, see “Perform Coredump of Failed Primary RP after Switchover Software Unit Functional Spec. (ENG-84351).”

24.4.2.9 Event Tracer

The Event Tracer subsystem is a general debugging subsystem to aid development engineers trace module flow and system events. It provides general hooks that can be inserted into the subsystem to be traced. The subsystem to be traced will provide general tracing filters. The hooks will then trace the events based on the filters supplied. The saved debug information can be extracted and then analyzed. This will help in keeping a log of the recent activities of the subsystem while maintaining a minimal overhead in doing so. It stores the data in a binary format without applying any formatting or processing. This accommodates tracing events that generate a lot of data very quickly. The traced data can also be optionally stored in a file. This file can later be moved to another location and different formatting can be applied to it for further analysis.

For more information, see section 20.8 “The Event Trace Facility.”

24.4.3 Platform Specific Support

There are a number of areas in which platform specific support must be implemented in order to support SSO on that platform. The following sections discuss these areas.

24.4.3.1 Line Card Drivers

Two mechanisms have been developed to synchronize interface state information from the Active RP to the Standby RP:

- 1 Platform-dependent device driver synchronization of line card events
- 2 Synchronization of interface state information by platform-independent code in the media layer code (a.k.a. network.c).

The first is described here and is used by the ESR platform implementation.

In this approach events received by the device drivers from the line cards are processed on the Active RP and then forwarded to the Standby RP essentially unchanged for processing there. The event is directed to the driver instance on the Standby RP, where it is processed in much the same way as on the Active RP. In processing the event, the Standby driver instance applies knowledge of the redundant system state as appropriate.

Depending on the type of event, the Standby driver may notify the platform-independent interface layer and layered protocols and features by invoking functions such as `net_cstate()`. In the case of an interface up/down state change, the standby will also update the IDB shadow state and invoke the `standby_hwidb_state_change()` registry.

24.4.3.2 RF

There is platform-dependent code required to tie RF into platform events. This is usually referred to as “porting RF” to a particular platform. RF has been ported to the ESR and is in the process of being ported to the GSR and 7500.

24.4.3.3 IPC

There is some platform-specific work necessary to support IPC. The driver code for the IPC media interface is platform specific. Some modifications may be needed there. In addition, the version negotiation needs to be performed in platform-dependent initialization code (although the algorithm is generic). These modifications will be done by the IOS Infra IPC development team for the initial platforms.

The other area that requires platform dependent IPC changes is for those platforms that have ported IPC to another OS that communicates with IOS IPC (e.g., LCDOS on the ESR Line Cards). These ported implementations have to be updated at some time to support the new features of IPC (IPC on IOS will remain backward compatible with the current implementations, so this is not an immediate requirement). This is not a first phase deliverable and is not considered further here.

24.4.3.4 APS Support

Since APS is supported on the different platforms in a platform specific way, there is some amount of work to do on each of the platforms to ensure APS works at least as well as it does on an RPR+ or predecessor IOS implementation. The ESR is planning on implementing a stateful client to support APS on that platform. The approach on the GSR and 7500 is still TBD.

24.4.3.5 ATM Drivers and Line Cards

Since ATM is supported on the different platforms drivers and Line Cards in a platform-specific way, there is some amount of work to do on each of the platforms to ensure that the stateful implementation of ATM works with the SSO ATM implementation.

In general, the ATM driver on the Standby should handle setupvc()/teardownvc() function vectors invoked from the ATM common layer to initialize the device instance specific structures and should avoid sending IPC message to the line cards.

For more information, see “IOS ATM HA Architecture and Software Functional Spec. (ENG-104679).”

24.4.3.6 Post Switchover State Verification and Reconciliation

After a switchover event occurs, the state information that exists on the Standby RP may not be completely accurate with respect to what was current on the Active RP at the time of the switchover. This might happen for several reasons including a synchronization message in progress on the Active not being delivered to the Standby because the Active failed at an inopportune time. Such a condition, left uncorrected, could result in customer surprise and incorrect system operation after the switchover. The reconciliation process performed after switchover is platform dependent and will likely be different on each platform due to different hardware implementations and considerations. The time taken to perform this reconciliation is critical as it directly affects the switchover time until the interfaces are available for both forwarding and for protocol processes to send keep-alives to their peers.

For more information about the ESR, see “Omega IOS Non-Stop Forwarding /Stateful Switchover SFS (ENG-83572).”

24.4.4 Network Management Support

Network management support must be provided for SNMP data and MIB variables appropriately on an SSO redundant system while taking care not to break existing usage by outside agents while still satisfying the requirements of platform implementations which refer to and use this data.

There are several MIB variables, some new, some existing, which we believe must exist for a redundant system. Discussion at the weekly HA Technical Meeting decided that there were at least four variables required, one of which already exists but needs both its meaning and behavior altered in an SSO system. Some would serve customer needs and some would be useful for internal monitoring and reporting:

- sysUpTime—System uptime represents the total time that this system has been providing uninterrupted service (from an external point of view). This value is incremented only on the Active RP. It is stateful and represents a continuously increasing time value. This is required for LMI to work properly for ATM.
- [new]—The RP uptime; the total time that this RP has been up without a failure; i.e., the time since the last boot until “now” on this RP.
- [new]—The time this “this” RP has continuously served as the Active RP.
- [new]—The convergence time the time between when this RP began forwarding as the Active RP and the time the routing protocols declared convergence (from CEF, who “knows” this).

Other network management issues are currently being worked on and are TBD.

24.5 Redundancy Facility

State synchronization is used by the IOS protocols and features which support HA to maintain sessions, circuits, and feature state through an RP switchover. At a low level, state synchronization involves copying state information from the active RP to the standby RP. State information received by the standby RP is distributed to the peer features and protocols for integration on the standby RP. State distribution may occur either as a result of changes in system state or on an ongoing basis.

In general, databases are synchronized, rather than raw data. This allows data abstraction and enables interoperation of different versions of hardware and software.

A centralized component provides an infrastructure for state synchronization and for construction of HA-capable clients generally. This function is the IOS Redundancy Facility, or RF. The primary function of the RF is to provide a platform-independent, abstracted set of services to client protocols and services which seek to support HA.

The IOS RF design is based on software developed for the Harley program, as described in “Harley Redundancy Framework Functional Spec. (ENG-47384).” Some changes in detail are motivated by differences in platform architecture and services, as well as a desire for some measure of compatibility with HSA / EHSA / RPR.

The RF design places functions into two major categories:

- RF Clients—Clients are entities which utilize the RF infrastructure to share state information between the two RPs. Clients may register with the RF for notification of significant redundancy events, including changes in redundancy control configuration and notification of system state progression. An IOS feature or protocol may implement one or more RF clients to support non-service-affecting switchover in the feature or protocol.
- RF Infrastructure—The RF infrastructure exports a variety of services to RF clients. It is important to note that the RF infrastructure does not itself provide state synchronization or non-service-affecting switchover - the actual state synchronization protocols are developed as part of client development. This complicates client development, but allows the infrastructure to be lightweight and modular, and therefore extensible.

24.5.1 RF Clients

RF clients maintain data base synchronization between their active and standby instances. As described, the format of client messages and the details of the synchronization protocol is client-specific. The timing and content of client synchronization must be done within the context of RF.

Associated with a client is its client identifier, a version, and a client sequence number. The client identifier uniquely identifies the client. When the client uses the RF peer messaging API, the client ID is placed in the redundancy messages exchanged with a peer instance, allowing RF message de-multiplexing.

The client version corresponds to the software and functional version of the client. Two versions of clients that differ from the perspective of peer instance are assigned a different version number. The version number is used to invoke version-specific message/data transformation functions internal to the receiving client. Versioning is a requirement for seamless software upgrades and downgrades.

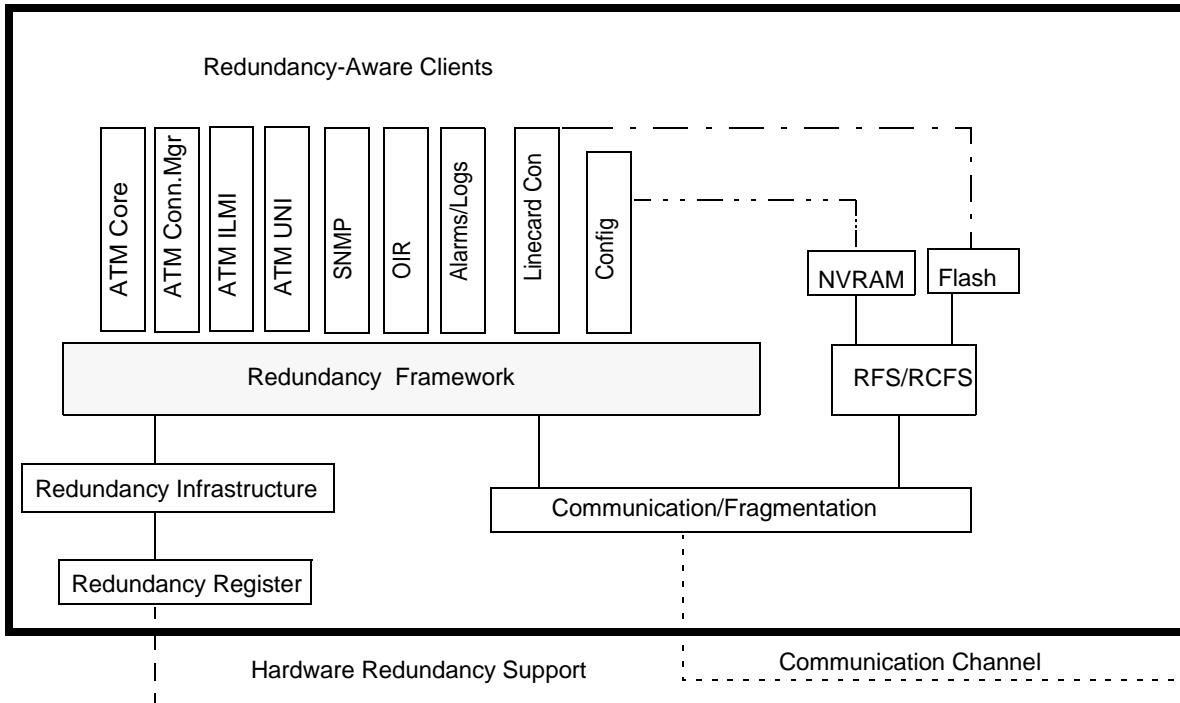
The client sequence number is used by RF to group clients as they register with the RF. When redundancy events occur registered client call-backs are invoked in order beginning with lowest sequence number. Clients which have duplicate redundancy sequence numbers will be invoked in the order in which they were registered. The redundancy sequence number is intended to help manage dependencies among clients.

A client may also register with RF for notification of system state progression and changes in redundancy control configuration. These notifications are provided by the RF in the form of synchronous and asynchronous callbacks. Clients are expected to use events to control data synchronization activity.

The design of client data extraction and integration routines is left to the client. These routines can be integrated into the existing protocol or feature, or can be designed as a new function that cooperates with the existing protocol or feature implementation. (This is important when making existing IOS applications redundant aware as compared to designing a new application with redundancy in mind.)

Figure 24-8 describes the RF software architecture.

Figure 24-8 Software Architecture of Redundancy Framework



24.5.2 RF Client Message Transport

The RF offers a data transport service to clients. A client may send synchronization messages to a peer and may also register with the service for delivery of messages sent by a peer.

The RF requires the host platform provide the underlying inter-unit message transport. The platform layer may be reliable or unreliable. An unreliable platform transport is preferred for simplicity, efficiency, and best performance. On the 7500, 6400, 10000, and Harley / 6100 NI-2 platforms the IOS IPC facility provides the transport layer. UDP/IP may also be used if the platform supports use of this protocol between units.

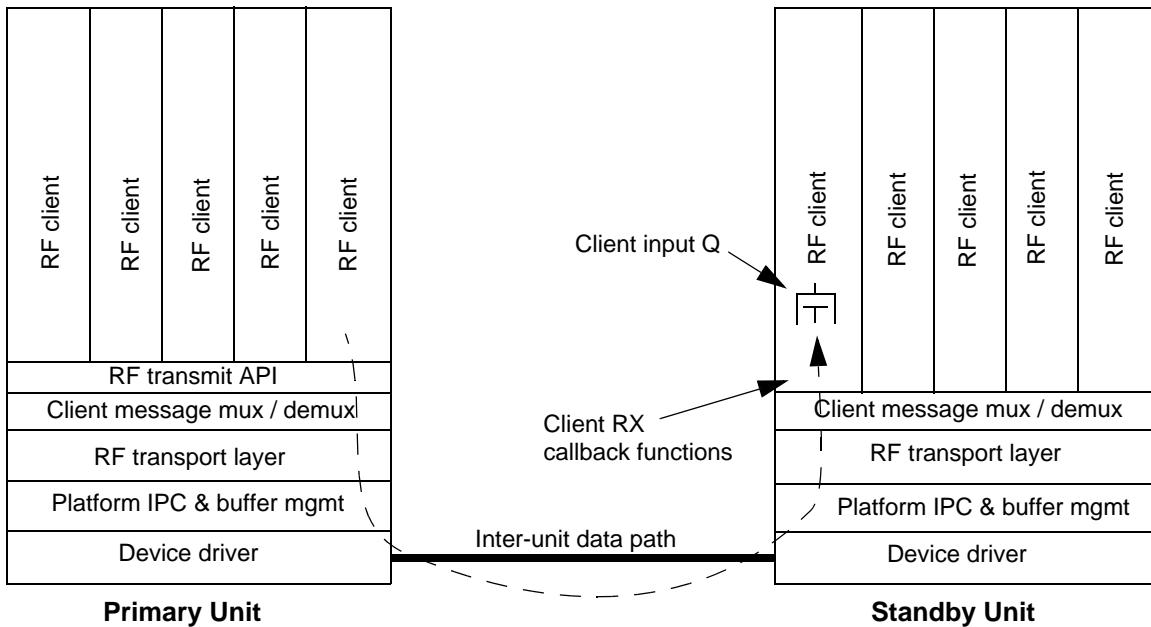
The RF requires the platform layer provide buffer management and message send interfaces. These are registered with the RF during system initialization. The registries involved are `reg_invoke_rf_open_peer_comm()`, `reg_invoke_rf_send_peer_msg_blocked()`, and `reg_invoke_rf_send_peer_msg()`.

The RF exports a receive notification interface to the platform-specific transport layer. This interface is called by the platform code when a message is received over the inter-unit data path. The registry is `reg_invoke_rf_route_peer_msg()`.

Client messages are prefixed by a common header. The header contains storage for a client ID, message sequence number, and client version. The client ID identifies the sending client and the intended recipient. The version field may be used to apply version-to-version data transforms prior to delivery to the client.

Figure 24-9 illustrates client message routing:

Figure 24-9 Client Message Routing



24.5.3 RF Infrastructure

The RF infrastructure provides a series of client services as well as functions for control and monitoring of system redundancy. Control over the gross behavior of the redundant RPs and clients is provided by the RF.

24.5.3.1 Redundancy Control and Monitoring

The RF provides functions and interfaces for control and monitoring of the platform's redundant features, including IOS CLI support for:

- Data synchronization enable / disable
- Monitoring the current system redundancy configuration and state
- Configuring the redundant system elements.
- Associating slots for SONET 1:1 and 1:N APS

Note On some platforms these three are provided by EHSA CLI and SONET APS CLI. We need to reconcile the existing redundancy control and Harley RF CLIs. Whether the code is physically integrated with the RF code is less important, but it should be thought of as a single logical entity.

- Manual RP switchover control
- Entry to and exit from split mode
- Manual SONET APS interface switchover control
- RF keepalive protocol control and configuration

- Platforms may disable the RF keepalive mechanism in favor of a platform-specific mechanism. Control is through a programmatic interface. RF monitors the ongoing keepalive process.
- Display of client lists and associated per-client information
- RF event and message debug
- Control over RP negotiation preference, if supported on the platform

A representative detailed CLI syntax description is available in section 24.6.9 “CLI Support.”

The RF instantiates and supports a MIB, which is defined in Appendix B in the “RF User’s Guide (ENG-71183).”

24.5.3.2 Fault Management & Switchover Notification Input

The RF links with a platform-specific function which integrates faults and produces a switchover decision. This function must be instantiated and is unique to each platform.¹

Platform code may also register a callback functions which will be invoked by the RF prior to a manually-initiated switchover and entry and exit from split mode.

24.5.4 Redundancy States

Redundancy Facility maintains several states. Transition into each state is presented to registered client callbacks. Clients can also determine local state and peer state via `reg_invoke_rf_get_my_state()` and `reg_invoke_rf_get_peer_state()`, respectively. Each registered client has a client ID and a client sequence number. Client sequence numbers represent the order in which clients are notified of the state transition. This order of event notification provides the basis of progression, state transition from the “lower” state to the “higher” state. Each client receives all event notifications through its registered callback. The client callback can choose to process the event or simply return. In either case, the client must return to RF with the proper return code.

If the client has no work to do when the progression event callback is invoked, it returns `RF_DONE`. If the client has significant work to do (i.e. must contact peer client and wait for a reply, etc., then the client returns `RF_OK`. At this point, RF will pause progression while the given client completes its processing. The client will have a certain amount of time in which to complete its processing. If it exceeds this amount of time, the “notification timer” will expire, and progression will be considered to have failed. The peer will be reloaded at this point.

If the client returns any code other than `RF_OK` or `RF_DONE` (which would indicate an error), progression will be considered to have failed, and the peer will be reloaded.

RF states are:

- 1** Disabled
- 2** Initialization
- 3** Standby-Negotiation

The following states are transitioned during the standby-hot progression.

- 4** Standby-Cold
- 5** Standby-Config
- 6** Standby-File System

1. It may be possible to develop common code for sharing among platforms.

7 Standby-Bulk

8 Standby-Hot

The following states are transitioned during the active progression.

9 Active-Fast

10 Active-Drain

11 Active Pre-Configuration

12 Active Post-Configuration

13 Active

24.5.4.1 Initialization

This state is used to establish any platform dependent capabilities required for redundancy. Examples may include establishment of platform communication facilities, such as IPC, or setting hardware to a neutral or predefined state. The processing required tends to be platform specific.

24.5.4.2 Standby-Negotiation

In this state, each unit determines if the peer unit is present and then negotiates for activity. The peer detection (see Table 24-1) and negotiation (see Table 24-2) processing is platform-specific. The platform routine, once negotiation has completed, reports to RF that the unit will be standby-cold or immediately transition to active (active progression).

The negotiation is accomplished by platform dependent routines, but would likely be built on the simplified state matrix as indicated in Table 24-1:

Table 24-1 Simplified Detection Matrix

	Primary unit not present	Primary unit present
Secondary unit not present	N/A	Unit immediately goes through active progression to become the active unit
Secondary unit present	Unit immediately goes through active progression to become the active unit	See Negotiation Matrix as indicated in Table 24-2.

Table 24-2 Simplified Negotiation Matrix

	Primary Unit Negotiating	Primary Unit Active
Secondary Unit Negotiating	Negotiation Clash, the Primary unit becomes active via active progression, and the secondary unit goes standby-cold.	Secondary unit goes standby-cold.
Secondary Unit Active	Primary unit goes standby-cold.	Negotiation clash. The primary unit remains active while the secondary unit must back-down. The Back-down occurs through a reload of the secondary unit. This is needed to correct data synchronization.

Note This is referred to as clash. In a loosely coupled system, the probability of clash is related to the amount of underlying h/w support. If all a system has is IPC or UDP, then the system will have a greater probability of clash. If a platform provides a second channel of h/w support, then this can be reduced or eliminated.

24.5.4.3 Standby-Cold

This state is entered when negotiation has determined that the peer unit is active. In this state the unit waits for the active unit to progress through the standby states. Progression through the standby states is initiated and controlled by the active unit.

24.5.4.4 Standby-Config

In this state the active unit updates the standby unit configurations, boot config and running config. Information flowing between the active client and the standby client is client specific.

24.5.4.5 Standby-File System

In this state the active unit updates the standby unit file system. This is platform and system dependent. Information flowing between the active client and the standby client is client specific.

For example, a single client could be responsible for sync'ing the file system, or files could be sync'ed by individual clients. The file system could be traditional disk or flash-based.

24.5.4.6 Standby-Bulk

In this state all redundancy aware subsystems in the standby unit are being updated with the context of the active unit in preparation to becoming active. This is the process by which redundancy clients transfer internal data from the active unit to its standby instance. Information flowing between the active client and the standby client is client-specific.

For example, the active ATM client would need to synchronize its established VCs to the standby ATM client.

24.5.4.7 Standby-Hot

In this state redundancy standby clients continue to receive data events from active clients. This process of dynamic data updates keeps standby clients up-to-date with the latest context changes of the active unit in preparation for a sudden switch of activity. Information flowing between the active client and the standby client is client-specific.

For example, active ATM client needs to send SVC established events and SVC teardown events to the standby ATM client. This insures that the standby unit can become active with the latest ATM fabric configuration.

24.5.4.8 Active-Fast

This is the first state of active progression. A transient state that is entered immediately after determining that the standby unit must become active. This provides an opportunity for clients to complete time critical platform dependent processing required to maintain calls.

24.5.4.9 Active-Drain

This is a transient state that clients can use to process queued messages or scrub data structures.

24.5.4.10 Active-Preconfig

This is a transient state that clients can use to prepare for system configuration. After all clients have finished processing for this state, the system configuration is then processed, and RF transitions to the next state.

24.5.4.11 Active-Postconfig

When this state is entered, system configuration has been completed and any desired processing such as configuration reconciliation may now be performed.

24.5.4.12 Active

This state indicates that the unit is fully active and is processing calls. Once the standby unit is recognized, the active unit will initiate standby progression to transition the standby unit from standby-cold to standby-hot. Once the standby unit reaches standby-hot state, the system is able to switch-activity and retain established calls.

24.5.4.13 Redundancy State Progression

The following RF progression events are passed to clients to coordinate RF state transitions and redundancy. Clients receive progression events through a registered progression call-back. The client call-back has the option of processing the event or simply ignoring it.

RF_PROG_INITIALIZATION

This progression event is always generated during system initialization. It is intended to help coordinate system or platform services that are needed by RF. For example, creating buffer pools or configuring IPC.

RF_PROG_STANDBY_COLD

When received, this progression event indicates that this unit is to become the Standby-Cold unit.

RF_PROG_STANDBY_CONFIG

On the active unit this progression event indicates that the active client should sync its configuration data to the standby unit.

RF_PROG_STANDBY_FILESYS

On the active unit this progression event indicates that the active client should sync its files to the standby unit file system.

RF_PROG_STANDBY_BULK

On the active unit this progression event indicates that the active client should sync its current context data as required.

RF_PROG_STANDBY_HOT

On the active unit this progression event indicates that the active client has successfully sync'd its context to the standby client and that the standby is now ready and able to gain activity and retain calls. This also implies that the active client should continue sending dynamic data events to the standby client.

RF_PROG_PLATFORM_SYNC

The active unit receives the RF_PROG_PLATFORM_SYNC event when the peer unit comes online for the first time. This event allows the platform to sync over any data needed on the inactive unit before it is allowed to continue initialization. For example, the active unit may need to sync EEPROM information to the standby unit.

RF_PROG_ACTIVE_FAST

This progression event is generated to standby clients immediately following the detection of an active unit failure. This notification begins the process of gaining activity, providing the earliest indication to clients that the standby unit is gaining activity. This notification is intended for platform dependent clients that need to manipulate hardware, such as taking control of the bus, to meet switch over requirements. This event is not generated on the (previously) active unit.

RF_PROG_ACTIVE_DRAIN

This progression event is generated such that clients can process queued messages, scrub data structures, etc.

RF_PROG_ACTIVE_PRECONFIG

This progression event prepares clients for system configuration.

RF_PROG_ACTIVE_POSTCONFIG

This progression event indicates that system configuration is now complete.

RF_PROG_ACTIVE

This progression event indicates that the unit is now fully active.

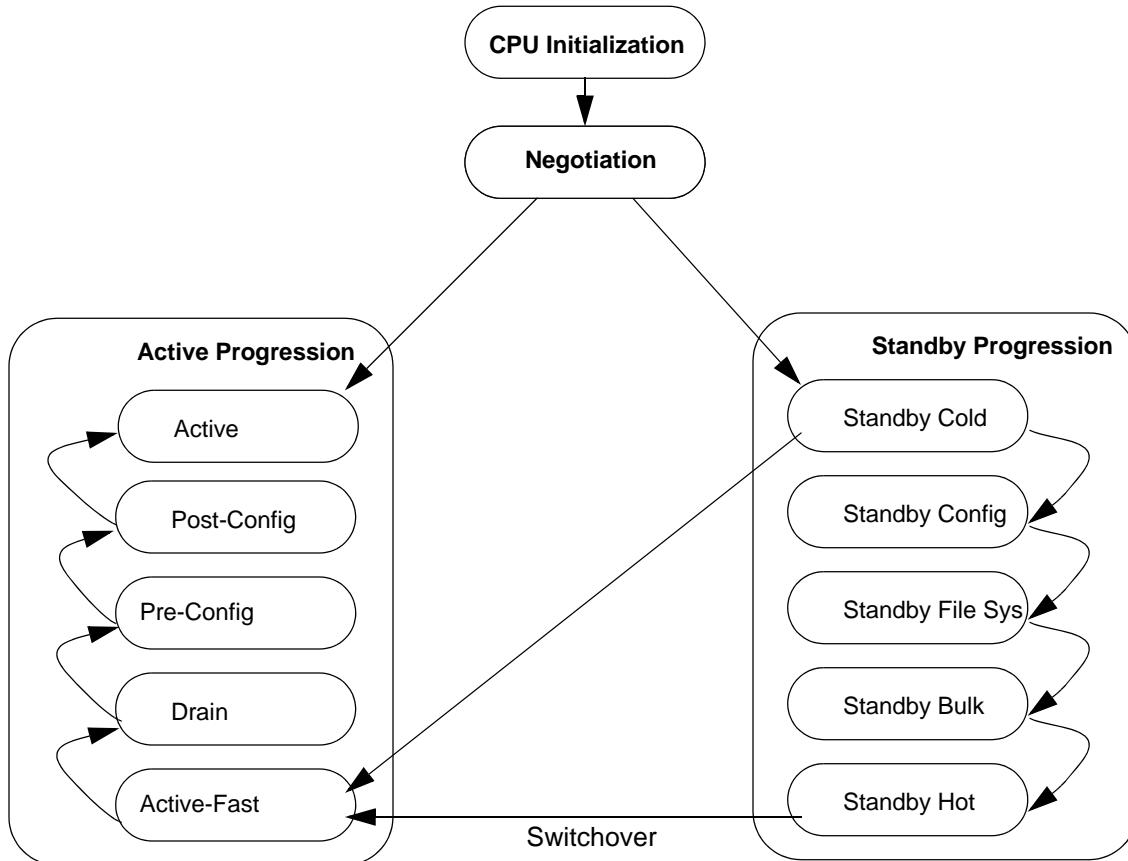
24.5.4.14 RF Progression

There are three phases to progression.

- 1 Initialization Progression
- 2 Standby Progression
- 3 Active Progression

Figure 24-10 illustrates the three phases of RF progression, beginning with CPU Initialization:

Figure 24-10 Three Phases of RF Progression



24.5.4.14.1 Initialization Progression

An RF unit will always go through initialization progression. This event should be used to complete platform dependent processing and setup RF required resources. Once complete, RF will transition to an internal negotiation state. So, initialization progression starts at unit power up (i.e. coming up from a reload command) to `RF_PROG_INITIALIZATION`.

The events received in this sequence (in order) are:

Step 1 `RF_PROG_INITIALIZATION`

24.5.4.14.2 Active Progression

Only one processor will go through the second progression sequence. i.e. only one processor can be active at any given time. So active progression starts with `RF_PROG_STANDBY_COLD` to `RF_PROG_ACTIVE`:

The events received in this sequence (in order) are:

Step 1 `RF_PROG_ACTIVE_FAST`

Step 2 `RF_PROG_ACTIVE_DRAIN`

- Step 3** RF_PROG_ACTIVE_PRECONFIG
- Step 4** RF_PROG_ACTIVE_POSTCONFIG
- Step 5** RF_PROG_ACTIVE

24.5.4.14.3 Standby Progression

The third progression sequence involves syncing data between the active processor and the standby processor. The ACTIVE processor which drives progression, will always receive the following progression messages (in order):

- Step 1** RF_PROG_STANDBY_CONFIG
- Step 2** RF_PROG_STANDBY_FILESYS
- Step 3** RF_PROG_STANDBY_BULK
- Step 4** RF_PROG_STANDBY_HOT

Whether or not the INACTIVE processor receives any messages depends on the return code the clients give RF. If a client progression call-back returns RF_DONE, RF assumes that this client had no progression work to do or all the progression work could be performed on the active unit. Therefore no progression sync message is sent to the inactive unit. If a client returns RF_OK, RF assumes that this client has progression work to do. So, an RF_EVENT_CLIENT_PROGRESSION will be sent to the same client on the inactive side, i.e. this is not a broadcast. Only the client that returned RF_OK will get this message. The operand parameter in the progression call-back will have the current progression state. So, if a client returns RF_OK when receiving the RF_PROG_STANDBY_CONFIG event, that same client on the inactive side would receive an RF_EVENT_CLIENT_PROGRESSION progression event with operand set to RF_PROG_STANDBY_CONFIG. After sending the inactive unit the RF_EVENT_CLIENT_PROGRESSION, RF will start a notification timer (30 secs). If the ACTIVE client does not call reg_invoke_rf_client_done before the timer expires, the active RF will consider the progression as failed and reload the peer. If the ACTIVE client does call reg_invoke_rf_client_done before the notification timer expires, progression will continue with the next client. If the ACTIVE client is still working on the sync and needs more time, it can extend the notification timer by calling rf_extend_notification_timer. A client can only extend the notification timer five times. So the client must complete its sync processing within.

Also the active unit will receive the RF_PROG_PLATFORM_SYNC event when the peer unit comes online for the first time. This event allows the platform to sync over any data it may need before the inactive unit is allowed to continue initialization.(MAC address for example).

24.5.4.15 RF Progression Synchronization

As the Active RF progresses, RF sends the event, RF_EVENT_CLIENT_PROGRESSION, to the standby RF with the current client ID. The standby RF uses the event to inform the specified standby client that it should prepare for the indicated progression.

RF_EVENT_CLIENT_PROGRESSION

This event is sent from the active RF to the inactive RF to coordinate the progression event with the respective client. For example, if an active client returns RF_OK when receiving the RF_PROG_STANDBY_CONFIG event, that same client on the inactive side would receive a RF_EVENT_CLIENT_PROGRESSION progression event with operand set to RF_STANDBY_CONFIG. This progression event is used to coordinate the active and inactive units through progression.

24.5.4.16 RF Progression Event Call-Back API

When clients register with RF, they provide a call-back to receive progression events. This function should process needed progression events and return the appropriate return code to RF. The call-back should minimize the event processing as the call-back is running under the (process) context of RF. If extensive processing is required, the call-back should post a message to the client task, or otherwise perform processing under its own thread.

If a client returns RF_DONE, RF assumes that this client had no progression work to do or all the progression work could be performed on the active unit. Therefore no progression message is sent to the inactive unit. If a client returns RF_OK, RF assumes that this client has progression work to do. In this case, the client must indicate to RF that it has completed processing the event by invoking `reg_invoke_rf_client_done`.

The prototype of the progression event call-back is:

```
typedef uint32 (*t_rf_progression_func)( uint32 clientID,
                                         uint32 clientSeq,
                                         uint32 rf_event,
                                         uint32 operand,
                                         uint32 rf_state,
                                         uint32 rf_peer_state );
```

Parameters:

- `clientID`—ID allocated to client
- `clientSeq`—client's notification sequence number
- `rf_event`—RF progression event being invoked
- `operand`—operand associated with the progression event (not always used)
- `rf_state`—current state of RF
- `rf_peer_state`—current RF state of the peer unit

24.5.5 Redundancy Status Events

RF status events are invoked to clients on both the active and standby units. Status events are synchronous call-backs that indicate specific conditions that may be of interest to clients.

RF_STATUS_PEER_PRESENCE

Indication that the presence of the peer unit has been detected (card inserted) or lost (card removed). When the system is in simplex mode clients should stop sending dynamic data updates.

RF_STATUS_PEER_COMM

Indication that the communication link to the peer unit has been lost or has been restored.

RF_STATUS_SWACT_ENABLE

Indication that the ability for the user to initiate a switch over has been enabled or disabled. A generous guard timer is started when CLI switch over is disabled. For example, the file system client would disable the ability to switch activity from the CLI when squeezing flash.

RF_STATUS_SPLIT_ENABLE

Indication that the standby unit has been logically disabled from the active unit with regard to sync'ing data. When the user disables split mode, the standby unit is reloaded. As the standby unit initializes, RF will go through the negotiation process, bringing the unit back to standby-cold (assuming the active unit did not fail during the time).

RF_STATUS_MANUAL_SWACT

Indication that the user (CLI or SNMP) has initiated a switch of activity. This is really a courtesy status that, depending upon the system, may not get distributed to all clients before the switch over.

24.5.5.1 RF Status Event Call-Back API

When clients register with RF, they provide call-back to receive status events. This function should process needed status events and ignore the others. The call-back should minimize the event processing as the call-back is running under the context of RF. If extensive processing is required, the call-back should format a message and send to itself.

The prototype of the status call-back is:

```
typedef uint32 (*t_rf_status_funct)( uint32 clientID,
                                      uint32 clientSeq,
                                      uint32 rf_status,
                                      uint32 operand,
                                      uint32 rf_state,
                                      uint32 rf_peer_state );
```

Parameters:

- `clientID`—ID allocated to client
- `clientSeq`—client's notification sequence number
- `rf_status`—RF status event being invoked
- `operand`—operand associated with the status event (not always used)
- `rf_state`—current state of RF
- `rf_peer_state`—current RF state of the peer unit

24.5.6 RF Messaging

The RF offers a data transport service to clients. A client may send synchronization messages to a peer and may also register with the service for delivery of messages sent by a peer.

The RF requires the host platform provide the underlying inter-unit message transport. The platform layer may be reliable or unreliable. An unreliable platform transport is preferred for simplicity, efficiency, and best performance. On the 7500, 6400, 10000, and Harley / 6100 NI-2 platforms the IOS IPC facility provides the transport layer. UDP/IP may also be used if the platform supports use of this protocol between units.

The RF requires the platform layer provide buffer management and message send interfaces. These are registered with the RF during system initialization.

The RF exports a receive notification interface to the platform-specific transport layer. This interface is called by the platform code when a message is received over the inter-unit data path.

Client messages are prefixed by a common header. The header contains storage for a client ID, message sequence number, and client version. The client ID identifies the sending client and the intended recipient. The version field may be used to apply version-to-version data transforms prior to delivery to the client.

24.5.6.1 RF Message Call-Back API

```
typedef uint32 (*t_rf_msg_funct)( uint32 clientID,
                                  uint32 clientSeq,
                                  uint32 rf_state,
                                  void *ptr2msg,
                                  void *ptr2data );
```

Parameters

- `clientID`—ID allocated to client
- `clientSeq`—client's notification sequence number
- `rf_state`—current state of RF
- `ptr2msg`—pointer to the message buffer (used for release)
- `ptr2data`—pointer to the first byte within the buffer to be used for client data

24.6 Sending Sync Messages

RF provides registries for sync buffer management and sync message send/receive. Clients can be designed to use the RF messaging capability or they can be designed to use another facility, such as IPC. While RF provides APIs to facilitate the sending and reception of sync messages, the underlying infrastructure is platform dependent. The platform should provide a dedicated free pool to be used for sync messages. This will protect the system against buffer starvation due to redundancy.

When a client sends a sync message through RF, RF immediately passes it to platform dependent I/O routine (IPC-R). RF does not queue messages. Message queueing is handled in the platform dependent I/O routine. Once the platform dependent IO routines (IPC-R) have sent the packet (all fragments), the original packet can be released.

Caution The platform provided I/O facility must provide fragmentation/reassembly if the transport frame cannot support the largest client packet.

Obviously, packets remain allocated for the entire time queued. This time will depend upon the queue depth and the service time to fragment and transmit the packet. Given there are a finite number of buffers, clients need to throttle messages to the peer unit. A client blasting messages could cause buffer starvation problem.

Caution The number of free pool buffers is an engineering effort dependent upon the number of redundant clients and the amount of data to be sync'ed and the dynamic data sync requirements and the platform I/O throughput. It is suggested that the platform support routines provide a separate buffer pool for sending of sync messages.

Caution Client protocols must be designed for S/W upgrade and downgrade. Please refer to ENG-80331, “Protocol Transformation.”

24.6.1 Client Message Considerations

Since buffers are finite resources, Clients must be designed to handle the case when no free buffers are available to send sync messages. This also implies that the standby client must be tolerant of not receiving every message. Several examples follow:

- During ATM SVC dynamic syncs, no free buffer could be allocated, so ATM must drop the sync, implying that the standby ATM never gets notification of the established SVC. The impact of this is situational.
- If a new call, SVC, was not synced, obviously the standby unit would not know about the call and could not maintain the call if the standby unit becomes active.
- It is also possible for the standby ATM to receive a SVC tear-down msg for a SVC that it does not know about. In this case, the standby ATM must be tolerant of the condition and cleanup as required.
- Another consideration is when standby ATM has received the SVC setup, but not the SVC tear down before the swact occurred. In this case, the newly active unit (once standby) must reconcile the ATM data structures and ATM fabric. This is platform specific.

It is not recommended that clients set timers to resync specific data when the first sync failed (no buffer available). If the system is so busy that all buffers are used, retries could actually make things worse. How long do you wait and for how many times should the client retry? The more time that transpires, the more likely that the sync message becomes stale.

Clients must be designed to tolerate sync message lost on the newly active unit. For when the previously active unit failed, sync messages queued in the active unit are lost. Thus newly active (previously standby) clients need to be tolerant of this fact and provide reconciliation as required when becoming active.

24.6.2 Header Files

24.6.2.1 RF Include Files for Clients

Client files should only include the following RF header files.

```
#include "rf_definitions.h"  
#include "rf_client_ID_list.h"  
#include "rf_registry.h"
```

24.6.2.2 RF Include Files for Supporting Platform Code

Platform dependent routines are required to support RF may need to access RF internal data structures. If this is required, the minimum list of RF header files would be:

```
#include "rf_definitions.h"  
#include "rf_client_ID_list.h"  
#include "rf_registry.h"  
#include "rf_db.h"  
#include "rf_priv_defs.h"
```

Sending Sync Messages**24.6.2.3 rf_client_ID_list.h**

This header file is used to statically define the client IDs and the client sequence numbers. RF Client IDs must be fixed, and the same between the active and the standby units so the two clients can communicate with each other. Client ID is used for all progression and messaging between units.

```
enum {
    RF_INTERNAL_MSG_ID,           /* RF reserved */
    C6100_RF_CLIENT_ID,
    TRM_CLIENT_ID,
    MTCE_RF_CLIENT_ID,
    OAM_RF_CLIENT_ID,
    EXAMPLE_1_CLIENT_ID,         /* use these for RF testing and regression */
    EXAMPLE_2_CLIENT_ID,
    RFS_CLIENT_ID,
    OIR_CLIENT_ID,
    APS_CLIENT_ID,
    C6100_SEC_CLI_ID,

    RF_LAST_CLIENT_ID,
};
```

Client sequence numbers determine the order that a client will be notified of RF events, relative to other clients. There are cases where one client must be notified before another. This should be noted when the sequence number is defined. The lower sequence numbers are notified first.

```
#define RF_INTERNAL_MSG_SEQ      ( 0 )
#define C6100_SMB0_CLI_SEQ       ( 5 )
#define C6100_CHASSIS_CLIENT_SEQ ( 7 )
#define C6100_RF_CLIENT_SEQ      ( 10 )
```

24.6.2.4 RF Message Header

For clients that choose to use the RF messaging APIs, the client must define its message formats such that the RF data structure is included in front of each message. RF uses this header information for message routing by client ID.

The `client_version` field should be used by clients to manage different sync protocol/message versions. If the client does not use the RF messaging APIs, the client message structure must include protocol versioning information. In either case, the client must be designed to support the current version and the two previous (major) versions. This is required to support software upgrades and downgrades.

```
typedef struct {
    uint32 clientID;
    uint32 clientSeq;
    uint16 length;           /* in bytes */
    uint16 client_version;  /* client sync protocol version control */
} tRF_SYNC_HEADER;
```

24.6.3 RF APIs

All RF APIs are defined in the `rf_registry.reg` file. The registry file is located in clearcase:

```
/vob/ios/sys/red_facility/rf_registry.reg.
```

The RF registry API is described in the “[High Availability](#)” chapter of the *Cisco IOS API Reference* and ENG-81946, the “RF Registry API” document.

24.6.4 RF History

RF maintains an internal history log of all significant RF events and status indications are logged, including:

- state transitions
- progression events
- internal events
- status events

Additionally, the following peer reload events are logged:

- “Reloading peer (notification timeout),”
- “Reloading peer (keepalive timeout),”
- “Reloading peer (this unit becoming active),”
- “Reloading peer (peer presence lost),”
- “Reloading peer (communication down),”
- “Reloading peer (CLI requested),”
- “Reloading peer (split mode cleared),”

The following are not logged:

- peer-to-peer messages
- data-syncs (aka keepalives)

24.6.5 Log Implementation Details

The log consists of “tokenized” records (i.e. not strings). Each record consumes 34-bytes. Records are dynamically allocated, so the log only consumes as much memory as is needed. It is capped at 500 entries, limiting it to a max of 16k. If it exceeds 500 records, the oldest records are purged from the log.

24.6.6 How Events Are Logged

The following four function calls, all internal to RF, are used to log events:

- **rf_log_state**(uint32 state, rf_history_type_e type);
Used to log my or peer state transitions. Non-transitions are filtered out as necessary.
- **rf_log_event**(uint32 event, uint32 client_id, uint32 client_seq, uint32 operand, uint32 rc, rf_history_type_e type);
Used to log internal, status and progression events.
- **rf_log_misc_event**(rf_history_type_e type);
Used to log miscellaneous events, like peer reload events.
- **rf_log_string**(const char *string);

Used to log runtime-generated text strings. These are currently limited to 20 characters to keep the overall log size down. Note that longer statically-indexed text strings can be logged with the previous function call.

These function calls are sprinkled throughout the RF subsystem. It was not possible to implement the history logging functionality as an RF client simply because not all relevant events could be captured in this manner.

24.6.7 History CLI

One additional CLI exec command was added to view the log:

show redundancy history

Additionally, the history log is included in “show techsupport” output to aid in debugging field calls.

The log history can also be cleared with the command:

clear redundancy history

24.6.7.1 Log Output - From the Active

```
Stinky#show redundancy history
Redundancy Facility Event Log:
00:00:39 *my state = INITIALIZATION(2) peer state = UNKNOWN(0)
00:00:39 RF_PROG_INITIALIZATION(0) RF_INTERNAL_MSG(0) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 Chassis Redundancy(15) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 RF_Client(4) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) MTCE RF Client(6) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) RFS client(10) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 OIR_Client(11) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) APS Client(12) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) RF_LAST_CLIENT(21) op=0 rc=11
00:00:39 *my state = NEGOTIATION(3) peer state = UNKNOWN(0)
00:00:49 RF_EVENT_GO_ACTIVE(31) RF_INTERNAL_MSG(0) op=0
00:00:49 *my state = ACTIVE-FAST(9) peer state = UNKNOWN(0)
...cut...
Aug 16 10:34:43 RF_EVENT_STANDBY_PROGRESSION(21) RF_INTERNAL_MSG(0) op=8 rc=0
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) RF_INTERNAL_MSG(0) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) C6100 Chassis Redundancy(15) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) C6100 RF_Client(4) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) MTCE RF Client(6) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) RFS client(10) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) C6100 OIR_Client(11) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) APS Client(12) op=0 rc=11
Aug 16 10:34:43 RF_PROG_STANDBY_HOT(5) RF_LAST_CLIENT(21) op=0 rc=11
Aug 16 10:34:51 my state = ACTIVE(11) *peer state = STANDBY READY(8)
```

24.6.7.2 Standby Logs (After a Switchover)

```
Smelly#show redundancy history
Redundancy Facility Event Log:
00:00:39 *my state = INITIALIZATION(2) peer state = UNKNOWN(0)
00:00:39 RF_PROG_INITIALIZATION(0) RF_INTERNAL_MSG(0) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 Chassis Redundancy(15) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 RF_Client(4) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) MTCE RF Client(6) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) RFS client(10) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 OIR_Client(11) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) APS Client(12) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) RF_LAST_CLIENT(21) op=0 rc=11
00:00:39 *my state = NEGOTIATION(3) peer state = UNKNOWN(0)
00:00:45 RF_STATUS_PEER_PRESENCE(14) RF_INTERNAL_MSG(0) op=1
00:00:45 RF_STATUS_PEER_PRESENCE(14) C6100 SMB0 Client(14) op=1
...cut...
00:05:19 *my state = ACTIVE(11) peer state = DISABLED(1)
00:05:19 RF_PROG_ACTIVE(8) RF_INTERNAL_MSG(0) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) C6100 Chassis Redundancy(15) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) C6100 RF_Client(4) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) MTCE RF Client(6) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) RFS client(10) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) C6100 OIR_Client(11) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) APS Client(12) op=0 rc=11
00:05:19 RF_PROG_ACTIVE(8) RF_LAST_CLIENT(21) op=0 rc=11
```

24.6.8 Log Timestamps

Each record includes a timestamp. This timestamp is one of two formats: uptime or a “real” syslog time. Uptime is always available but is more difficult to interpret as days and months pass. And syslog time is not available until it is manually configured or automatically set via NTP. Neither of these occur until after config time, so for at least a portion of the time, only uptime is available. So, logging uses uptime until syslog time is available. An entry is added to the log correlating syslog time and uptime so that a delta time spanning the timestamp transition can still be computed. The following log segment shows the transition from uptime to syslog timestamps.

```
00:00:49 RF_PROG_ACTIVE(8) RF_LAST_CLIENT(21) op=0 rc=11
Aug 16 08:46:23 Changing to system clock timestamps at uptime 00:23:15
Aug 16 08:46:23 RF_STATUS_PEER_PRESENCE(14) RF_INTERNAL_MSG(0) op=1
```

24.6.8.1 syslog Timestamps

If you need syslog timestamps, the following should be added to the IOS config:

```
service timestamps log datetime localtime
clock timezone EST -5
clock summer-time EST date Apr 1 2000 21:00 Oct 28 2000 21:00
ntp source Ethernet0/0
ntp server 171.69.6.2
ntp server 171.69.6.1
```

24.6.9 CLI Support

24.6.9.1 Exec Level Commands

The following exec level commands have been provided for CLI support for RF:

- **redundancy switch-over [force]**

This command causes a manual switch of activity to occur. It must be issued on the active unit, and will take effect if RF is in a state to allow switchover (the standby is in “Standby Hot” and no client or the platform code is temporarily disallowing switchover). If RF is not in a state to allow switchover, the “force” option may be used to cause an unconditional swact. Under normal conditions this should never be done, as it may bring up the standby in an unstable state. In fact, the standby unit may recognize this instability and reload itself to prevent instability.

- **redundancy reload peer**

This command reloads the peer via platform-dependent (usually hardware) means.

- **redundancy reload shelf**

This command reloads the peer (via a platform-dependent means) and also reloads the unit on which the command was entered.

24.6.9.2 Show Commands

The following show commands have been provided for CLI support for RF:

- **show redundancy**

```
Ralph-cpld8.1#show redundancy ?
clients    Redundancy Facility (RF) client list
counters   Redundancy Facility (RF) operational measurements
history    Redundancy Facility (RF) history
states     Redundancy Facility (RF) states
```

- **show redundancy clients**

```
Ralph-cpld8.1#show redundancy clients
clientID = 0      clientSeq = 0          RF_INTERNAL_MSG
clientID = 4      clientSeq = 10         C6100 RF_Client
clientID = 8      clientSeq = 100        RF_Example-ONCE
clientID = 10     clientSeq = 605       RFS client
clientID = 11     clientSeq = 610       C6100 OIR_Client
clientID = 12     clientSeq = 615       APS Client
clientID = 9      clientSeq = 901       RF_Example-TWICE
clientID = 14     clientSeq = 9999      RF_LAST_CLIENT
```

- **show redundancy counters**

```
Ralph-cpld8.1#show redundancy counters
Redundancy Facility OMS
    comm link up = 0
    comm link down down = 0
    invalid client sends = 0
    null sends by client = 0
        send failures = 0
    send msg length invalid = 0
    Client not rcving msgs = 0
    rcv peer msg routing error = 0
        null peer msg rcvs = 0
        erred peer msg rcvs = 0
            buffer gets = 90
    no buffers available = 0
        rls buffer count = 90
    buffer rls buffer errors = 0
    duplicate client registers = 1
    failed to register client = 0
    Invalid client syncs = 0
```

- **show redundancy states**

```
Ralph-cpld8.1#show redundancy states
    my state = 11 -ACTIVE
    peer state = 1 -DISABLED
        Mode = Simplex
        Unit = Primary

    Split Mode = Disabled
    Manual Swact = Disabled Reason: Simplex mode
    Communications = Down Reason: Simplex mode
    client count = 8
    client_notification_TMR = 30000 milliseconds
        keep_alive TMR = 2000 milliseconds
        keep_alive count = 0
        keep_alive threshold = 7
    RF debug mask = 0x0
```

- **show redundancy history**

```
Stinky#show redundancy history
Redundancy Facility Event Log:
00:00:39 *my state = INITIALIZATION(2) peer state = UNKNOWN(0)
00:00:39 RF_PROG_INITIALIZATION(0) RF_INTERNAL_MSG(0) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 Chassis Redundancy(15) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 RF_Client(4) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) MTCE RF Client(6) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) RFS client(10) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) C6100 OIR_Client(11) op=0 rc=11
00:00:39 RF_PROG_INITIALIZATION(0) APS Client(12) op=0 rc=11
```

24.6.9.3 Debug CLI

RF provides surgical debugging. That is, the ability to debug only the specific areas necessary, as compared to enabling all RF debug capability.

- **debug redundancy**

```
Ralph-cpld8.1#debug redundancy ?
      ehsa      Redundancy Facility (RF) EHSA
      fsm       Redundancy Facility (RF) FSM events
      kpa       Redundancy Facility (RF) keep alive
      msg       Redundancy Facility (RF) Messaging events
      progression Redundancy Facility (RF) Progression events
      status    Redundancy Facility (RF) Status events
      timer     Redundancy Facility (RF) Timer events
      errors    Redundancy Facility (RF) errors
```

- **show debug**

```
Ralph-cpld8.1#show debug
      Redundancy Facility FSM debugging is on
      Redundancy Facility timer debugging is on
      Redundancy Facility message debugging is on
```

24.6.9.4 Clear CLI

This command will clear the redundancy history log. It may be useful for debugging purposes.

- **clear redundancy history**

24.6.9.5 Configure CLI

RF provides a CLI sub-mode for its configure menu.

- **Redundancy CLI sub mode**

```
Ralph-cpld8.1(config)#redundancy

Ralph-cpld8.1(config.redundancy) ?
Redundancy configuration subcommands:
  (hidden) kpa-threshold      Specify Redundancy keep alive threshold
  (hidden) kpa-timer          Specify Redundancy keep alive timer
  (milliseconds)
    (hidden) notification-timer  Specify Redundancy notification timer
  (milliseconds)
    split-mode                Enable/Disable split mode
```

24.6.10 Platform-Dependent Support Routines

24.6.10.1 Platform Buffer Management

Platforms must provide call-backs to get and release buffers used for peer messaging and for RF control messages. Control messaging is used for internal RF events and interactions with the CLI and timers. Peer messaging are the buffers used by application when sending/receiving a sync message.

When practical, it is advisable that the peer messaging buffer pool be a separate pool. This is recommended so data sync messaging does not cause buffer starvation across the system. For client messaging, clients must throttle data sync traffic so not to starve RF or other clients. Clients must also handle the case of not getting a free buffer.

Below are example routines to get and release a RF control buffer.

```
/*
Syntax
uint32
rfpd_get_free_buffer( uint32 size_n_bytes, void **msg_ptr, void **data_ptr )

Description
get a free RF control buffer
Parameters
size_n_bytes      buffer size required - not used
**msg_ptr          pointer to msg to be released
**data_ptr         pointer to client application area
Returns
RF_OK             free buffer allocated to user
RF_INVALID_REQ    msg_ptr is not valid
RF_NO_RESOURCES   no buffer available
*/
uint32 rfpd_get_free_buffer (uint32 size_n_bytes, void **msg_ptr, void **data_ptr)
{
    paktype *ptr2pak;

    if( msg_ptr == NULL || data_ptr == NULL ) {
        return ( RF_INVALID_REQ );
    }

    ptr2pak = getbuffer( size_n_bytes );
    if( ptr2pak == NULL ) {
        *msg_ptr = NULL;
        *data_ptr = NULL;

        rf_oms.om_get_buf_no_resources++;
        return ( RF_NO_RESOURCES );
    }

    rf_oms.om_get_buf_count++;

    ptr2pak->application_start = ptr2pak->datagramstart;
    *msg_ptr = (void *)ptr2pak;
    *data_ptr = (void *)ptr2pak->application_start;

    return ( RF_OK );
}

/*
Syntax
uint32
rfpd_release_buffer( void *msg_ptr )

Description
release a RF control buffer back to its free pool
Parameters
msg_ptr           pointer to msg to be released
Returns
```

Sending Sync Messages

```

RF_OK           buffer released
RF_BAD_MESSAGE NULL buffer pointer
*/
uint32 rfpd_release_buffer (void *msg_ptr)
{
    paktype *ptr2pak;

    if( msg_ptr == NULL ) {
        rf_oms.om_rls_buf_errors++;
        return ( RF_BAD_MESSAGE );
    }

    ptr2pak = (paktype *)msg_ptr;
    datagram_done( ptr2pak );
    rf_oms.om_rls_buf_count++;
    return ( RF_OK );
}

```

24.6.10.1.1 Buffer Routine Registration

Below are code segments that register the same get/free routines for both the peer and control buffer management registries.

```

/*
 * Functions to get/release buffers for the peer queue
 */
reg_add_rf_get_peer_buffer(rfpd_get_peer_buffer, "rfpd_get_peer_buffer");
reg_add_rf_release_peer_buffer(rfpd_release_peer_buffer, "rfpd_release_peer_buffer");

/*
 * Functions to get/release buffers for the control queue
 */
reg_add_rf_get_free_buffer(rfpd_get_free_buffer, "rfpd_get_free_buffer");
reg_add_rf_release_buffer(rfpd_release_buffer, "rfpd_release_buffer");

```

24.6.10.2 Peer Messaging

Active-standby messaging facilities are platform specific. One platform may have IP capabilities, another ATM and yet another serial link. To support the many messaging options, RF provides registries for platform dependent facilities to send blocking, and send non-blocking peer messaging APIs. The platform must register these routines with RF. Clients should use the peer buffer routines for buffer allocation and buffer formatting.

24.6.10.2.1 Peer Messaging Registration

The following two platform functions will typically be wrappers to an underlying transport mechanism, and are used to transmit messages to the peer redundant system. The following examples use IPC.

```

/*
 * add the communication service function to be invoked when a
 * client sends a message to the peer unit.
 */
reg_add_rf_send_peer_msg (rfpd_send_peer_msg, "rfpd_send_peer_msg");

/*+*****+

```

Syntax

```
uint32
rfpd_send_peer_msg( void *ptr2msg )
```

Description

This is a platform dependent function that registers with RF. It is invoked to send msgs to the peer unit.

Clients are responsible for releasing buffers on send errors. This allows them to keep the buffer around in case of a retry. RF will release the buffer if the send is successful.

It is required that all clients include this header in the front of all sync messages sent to the peer unit via rf_send_msg_to_peer.

Clients are responsible for populating these fields before calling rf_send_msg_to_peer.

The client_version field should be used by clients to manage different versions of their sync protocol/messages. This is specifically important when the standby unit is being updated with a new version of s/w that has modified the sync proto.

```
typedef struct {
    uint32 clientID;
    uint32 clientSeq;
    uint16 length;           --- in bytes
    uint16 client_version;  --- client sync protocol version control
} tRF_SYNC_HEADER;
```

Parameters

*ptr2msg *This pointer must have been gotten by a call to rf_get_peer_buffer*

Returns

```
RF_OK
RF_INVALID_REQ      communication path is down or invalid,
RF_BAD_MESSAGE     invalid msg, such as NULL ptr
RF_TRANSMIT_ERROR   ipc send failed
*-----*/
```

```
uint32 rfpd_send_peer_msg (void *ptr2msg)
```

```
{
    ipc_error_code rc;

    /*
     * If the peer communication channel is down then do not let
     * clients send msgs to peer.
     */
    if (rf_db.comm_state != RF_TRUE) {
        rf_oms.om_invalid_sends++;
        return (RF_INVALID_REQ);
    }

    /*
     * NULL, nothing to do...
     */
    if (ptr2msg == NULL) {
        rf_oms.om_send_null++;
        return (RF_BAD_MESSAGE);
    }
}
```

Sending Sync Messages

```

rc = ipc_send_message((ipc_message *)ptr2msg, peer_ni2_port);
if (rc != IPC_OK) {
    rf_oms.om_send_failures++;
    return (RF_TRANSMIT_ERROR);
}
return (RF_OK);
}

/*
 * add the communication service function to be invoked when a
 * client sends a message to the peer unit. This version blocks
 * until an ACK is received from the peer unit.
 */
reg_add_rf_send_peer_msg_blocked(rfpd_send_peer_msg_blocked, "rfpd_send_peer_msg_blocked");

```

Platform code will also handle reception of messages via the underlying transport mechanism. It will then invoke `reg_invoke_rf_route_peer_msg()`, to route the message to the proper client, as in the following example:

Syntax

```
void
rfpd_rcv_peer_msg( ipc_message *ptr2msg, void *context, ipc_error_code rc )
```

Description

This is a platform dependent function that registers with IPC-R. It is called when a message from the peer unit has arrived.

Client is responsible for releasing the buffer provided RF can give the buffer to a client. If not, RF will release the buffer.

Parameters

<code>ptr2msg</code>	<i>pointer to the IPC-R msg rcvd</i>
<code>context</code>	<i>a copy of the context passed in when the function registered with IPC-R</i>
<code>rc</code>	<i>ipc_error_code</i>

Returns

None

```
void rfpd_rcv_peer_msg (ipc_message *ptr2msg, void *context,
                        ipc_error_code rc)
{
    tRF_SYNC_HEADER *ptr2rfheader;

    /*
     * Check the message
     */
    if (rc != IPC_OK) {
        rfpd_release_peer_buffer(ptr2msg);
        buginf("\nRcv peer msg: error %d in getting message.\n", rc);
        rf_oms.om_rcv_err++;
        return;
    }
    ptr2rfheader = (tRF_SYNC_HEADER *)ptr2msg->data;
```

```

reg_invoke_rf_route_peer_msg(ptr2msg, ptr2rfheader);

return;
}

```

24.6.10.2.2 Open Peer Communication Registration

```

/*
 * Add the function to open the RF IPC-R port to the
 * peer unit.
 */
reg_add_rf_open_peer_comm(rfpd_open_peer_comm, "rfpd_open_peer_comm");

```

24.6.10.3 OK to SWACT

RF CLI provides a hook for a platform-dependent routine which can be used to influence the processing of the **swact** command. This platform hook is not invoked in the RF force **swact** CLI command.

```

/*
Synopsis
static uint32
rfpd_OK_to_swact( uint32 my_state, uint32 peer_state )
Description:
A platform dependent function that can be used to allow or
disallow a manual swact. Based upon the retrun code, RF will
not swact, swact only if the user confirms it,
Parms
my_state
peer_state
Returns
RF_NOT_OK_TO_SWACT      Do not allow manual swact
RF_FORCED_SWACT_OK      Only swact if user confirms
RF_OK_TO_SWACT           swact away
*/
static uint32 rfpd_OK_to_swact (uint32 my_state, uint32 peer_state)
{
    /* add hooks into the system to determine if a swact should be allowed. */
    return (RF_OK_TO_SWACT);
}

```

24.6.10.4 OK to SWACT Registration

```
reg_add_rf_OK_to_swact(rfpd_OK_to_swact, "rfpd_OK_to_swact");
```

24.6.10.5 OK to Split

RF CLI provides a platform-dependent routine which can be used to check whether it is OK to enable or disable split mode It does not do the actual enabling/disabling.

```

/*
Synopsis
uint32

```

Sending Sync Messages

*rfpd_OK_to_split**Description:**RF Registered callback.**Checks whether or not it is OK to enable or disable split mode. It does not do the actual enabling/disabling.**Parms**rf_state
peer_state**Returns*

```
RF_NOT_OK_TO_SPLIT Do not allow it
RF_OK_TO_SPLIT Allow it
*/
uint32 rfpd_OK_to_split (boolean rf_split_indication,
                         uint32 rf_state,
                         uint32 peer_state)
{
    ...
    return (RF_OK_TO_SPLIT);
}
```

24.6.10.6 OK to Split Registration

```
reg_add_rf_OK_to_split(rfpd_OK_to_split, "rfpd_OK_to_split");

/*
 * Add the function to open the RF IPC-R port to the
 * peer unit.
 */
reg_add_rf_open_peer_comm( rfpd_open_peer_comm, "rfpd_open_peer_comm" );
```

24.6.10.7 Standby Initialization

The default behavior of the standby initialization is for it to suspend in `os/init.c` just before configuration is parsed. However, some redundant architectures may choose to fully configure the standby processor. RF allows for this. The platform would register for the RF API `reg_invoke_rf_extended_platform_init()`. This function is invoked when the standby unit is initializing and calls `reg_invoke_extended_ehsa_init()`. The platform code can select from the following behaviors as follows:

- 1 Return immediately with `rc = RF_START_PROGRESSION_NOW_SUSPEND_UNTIL_SWACT`. RF will start progression and will suspend until a switch of activity occurs. This is the default behavior.
- 2 Return immediately with `rc = RF_START_PROGRESSION_NOW_NO_SUSPEND`. RF will start progression and will not suspend.

Note With this option, there is a chance that the active will reset the peer since the progression messages sent by active may not be processed by standby until init is complete. This will result in notification timeout causing the active to reset the standby.

- 3 A variation on “2” would be for platform code NOT to return immediately, but instead to suspend until the desired platform-specified event(s) occur(s), and then return with `rc = RF_START_PROGRESSION_NOW_NO_SUSPEND`.
- 4 Return immediately with `rc = RF_START_PROGRESSION_ON_SYSTEM_INIT_NO_SUSPEND`. RF will start progression only when system initialization is complete (i.e. RF would do the equivalent of wait for the `systeminit_flag` becomes TRUE and then send the start message from the standby to the active). No suspension of `os/init.c` occurs for this option.
- 5 Return immediately with `rc = RF_START_PROGRESSION_NOW_THEN_CONFIG_SYSTEM`. RF will start progression immediately and suspend until progression is complete. At this point, `os/init.c` is resumed, which allows the standby to configure and initialize.

24.6.10.8 Process Configuration

For redundant systems that will suspend the standby unit initialization sequence prior to IOS configuration being read until a switch of activity occurs, RF provides a set of registries to control configuration processing. For redundant systems in which the standby unit will “come up fully” and parse its IOS configuration, the following registries do not need to be implemented.

- Specifically, on the unit becoming active, either as a result of bootup negotiation or as a result of a switch of activity, the following sequence of events will occur:
 - RF transitions to `RF_ACTIVE_PRECONFIG` and issues the corresponding event to all clients
 - RF calls `reg_invoke_rf_determine_config()`

The platform code can register for this function call and use it as a trigger to determine the location of the configuration which IOS will parse. For example, on the “active from boot” unit, the configuration may come from NVRAM, but on a system becoming active as a result of a switch of activity, the configuration may come from a memory buffer containing a configuration that was previously synced over from the peer before the swact occurred.

- RF calls `reg_invoke_rf_resume_extended_initialization()`

This releases the IOS initialization code from the second suspension point (point at which IOS init code invoked `reg_invoke_extended_ehsa_init()`), which allows the parsing of the IOS configuration to commence, and for IOS system initialization to complete.
- RF calls `reg_invoke_rf_config_post_processing()`

The platform code can register for this function call and use it to indicate that IOS parsing is complete and that the file containing the config (if it was not NVRAM) to be closed, etc.
- RF transitions to `RF_ACTIVE_POSTCONFIG` and issues the corresponding event to all clients

24.6.10.9 Reload

RF provides registries for invoking a reload of this unit, the peer unit and the shelf. These routines are platform-dependent. It is strongly suggested that the platform provide hardware support to reset the peer unit. This is required so the standby unit can reset a run-away active unit.

RF IOS CLI at this time does not provide a command to reload self. Reloading one self is done through the standard IOS **reload** command. Non-IOS platforms may use this registry.

24.6.10.9.1 Reload Registration

```
/*
 * Add platform dependent peer reset and shelf reload routines
 */
/*- reg_add_rf_reload_self(rfpd_reload_self, "rfpd_reload_self"); -*/
reg_add_rf_reload_peer(c6100_reload_peer, "c6100_rf_reload_peer");
reg_add_rf_reload_shelf(c6100_reload_shelf, "c6100_reload_shelf");
```

24.6.10.10 Primary Unit

In loosely coupled redundant systems there is a need to designate one unit the primary unit and the other as the secondary unit. The primary unit is the unit that becomes active when there is a negotiation clash. The ability to quickly resolve negotiation clash is an important capability.

```
/*
 * now we need to declare the primary-secondary unit.
 */
if( platform routine to determine if this is the primary unit ) {
    reg_invoke_rf_set_primary_mode(TRUE);
} else {
    reg_invoke_rf_set_primary_mode(FALSE);
}
```

24.6.11 RF MIB

SNMP agent code operating in conjunction with the CISCO-RF MIB enables a standardized, SNMP-based approach to be used in managing the RF features in Cisco IOS.

The SNMP objects defined in the CISCO-RF MIB can be viewed by any standard SNMP utility. The MIB can be referenced from Appendix B in the “RF User’s Guide (ENG-71183).”

24.7 Target Platform Architectures

The Cisco 7500, 12000 GSR, and 10000 ESR support centralized software-based forwarding, distributed software-based forwarding, distributed hardware-based forwarding, and centralized hardware-based forwarding respectively to meet the requirements of their target applications. The different architectures result in systems of varying software complexity. The following sections provide an overview and taxonomy of their respective architectures, including details of their forwarding and control architectures.

24.7.1 Cisco 10000 ESR

The Cisco 10000 ESR is initially targeted to high-density, low- and medium-speed aggregation applications. The ESR supports centralized, hardware-based forwarding under the control of a single IOS instance. The microcoded PXF forwarding engine is physically, logically, and electrically isolated from the line cards and from the embedded computer which runs the IOS control software.

ESR line cards include local control microprocessors which run a small, simple, embedded kernel and layered line card control code. The kernel, developed in-house, provides a very simple, non-preemptive, reliable execution model to the line card control code. Line card control code is specific to the line card type and is written for execution in the ESR line card environment, but is sometimes leveraged from existing, proven IOS interface control software.

Line card control processors and software are not involved in packet forwarding and contain no protocol processing code. The line card control software and IOS communicate using a message passing API.

ESR RP IOS is modified to remove unnecessary features and functions unrelated to the edge aggregation application. On the ESR, IOS does not forward transit packets and is protected from control plane overload by various mechanisms. A software fault on a line card or in the PXF forwarder is contained to the line card or to the PXF forwarder. Such faults are recovered locally and do not affect the stability of the IOS control plane.

The resulting distributed software system is relatively simple and maintains a high degree of separation between the control and forwarding functions.

Table 24-3 provides a simplified taxonomy of the architectural elements of the Cisco 10000 ESR:

Table 24-3 Cisco 10000 Edge Services Router

Architectural Element	Description
System	Protocols:IP Separate control & data planes: Yes, separate RP and FP on same card FP - Line Card Interconnect:
Multiple independent point-to-point links	One link per line card.
Direct LC-LC I/O	No
Standby RP/FP access to interconnect & line cards	Yes, loopback only RP - Line Card Interconnect: Shared Ethernet No maintenance bus
Route Processor	Has NME: Yes Redundancy: Yes, per RPR, RPR+
Forwarding Processor	Single, centralized, implemented by custom ASICs + microcode, atomic with RP Buffering: Centralized FP - Route Processor Interconnect: On-board PCIbus Software: Microcode bundled with IOS
Line Cards	Modular media interface (port adapters): No Redundancy: <ul style="list-style-type: none"> • 1:1: Yes, SONET APS for SONET cards, with external mux • 1:N: Yes, with external mux, not supported now Software: Line card LCDOS image bundled with RP IOS

24.7.1.1 Cisco 10000 ESR Forwarding and Control Architecture

The 10000 ESR is optimized for high-performance, high-density leased-line aggregation.

The system uses centralized forwarding; line cards are not substantially involved in the packet forwarding task. Forwarding is performed by the Performance Routing Engine, or PRE.

Forwarding is distributed to a microcoded forwarding processor, or FP; IOS is not directly involved in transit packet forwarding.

IOS runs on a separate route processor, or RP. The system supports two PREs. The PREs are atomic in the sense that switchover involves both the FP and the RP.

Control processors present on the line cards run a lightweight embedded kernel (LCDOS). The line cards maintain configuration and state information downloaded by IOS and developed locally. To avoid the need to reset line cards following an RPR/RPR+ switchover of the RP, all 10000 ESR line cards support a “soft reset” operation. The “soft reset” resets card parameters to default values and resynchronizes the state information maintained by the line card and by IOS.

For purposes of packet forwarding the line cards and the PRE are connected by a point-to-point interconnect known as the Iron Bus. No shared memory is present in the packet forwarding path.

A backplane Ethernet connects the RP with the line cards. Control traffic flows over this path.

24.7.2 Cisco 7500 Overview

Over time the Cisco 7500 router has been deployed as both a core router and an aggregation router in service provider networks. Today, the 7500 is deployed as a low-speed edge aggregation router in the IP Common Backbone network. In both applications the 7500 series has proven extremely flexible and robust.

The 7500, when configured with VIP cards, supports distributed, software-based forwarding of transit traffic. Forwarding is performed by the IOS image or images running on the VIP card(s) which terminate the input and output interfaces. The VIP IOS image is tailored for operation on the VIP and as such is functionally simpler than the IOS image running on the RSP.

In addition to forwarding packets, the VIP IOS image implements a variety of VIP hardware and distributed forwarding feature control functions. The VIP IOS image applies configuration and forwarding control parameters (for example, the CEF database) downloaded to it by the RSP IOS image. In so doing the VIP IOS image and the underlying single microprocessor combine the control and data planes.

There is one VIP IOS image per VIP card. In a 7513 system fully populated with VIP cards 10 or more separate IOS instances may exist in the chassis.

In the case in which a particular forwarding feature is not distributed on VIP cards, or in the case legacy Interface Processors rather than VIP cards are installed in the system, packets are directed to the RSP and forwarded by IOS running on the RSP. When performing this centralized, software forwarding task the RSP combines the control and data planes.

Table 24-4 provides a taxonomy of the architectural elements of the Cisco 7500 Multiprotocol router:

Table 24-4 Cisco 7500 Multiprotocol Router

Architectural Element	Description
System	Protocols: Many, including IP <ul style="list-style-type: none"> • Separate control & data planes: No, VIP CPU forwards & controls • Interconnect: Proprietary shared bus (CyBus) and packet memory (MEMD) Dbus maintenance bus
Direct LC-LC I/O	Yes, via MEMD
Standby RP/FP access to interconnect & line cards	No
Route Processor	Has NME: RSP2, RSP4: No, RSP8: Yes Redundancy: Yes, per HSA, RPR, RPR+

Table 24-4 Cisco 7500 Multiprotocol Router (continued)

Architectural Element	Description
Forwarding Processor	Centralized, combined with RP (for nondistributed features); IOS software-based Distributed, combined with line card CPU (for distributed features); IOS software-based Buffering: <ul style="list-style-type: none">• Distributed input and output on line cards• Centralized (MEMD) on RP and for LC-LC transfers
Line Cards	Modular media interface (port adapters): Yes Redundancy <ul style="list-style-type: none">• 1:1: Yes, SONET APS for SONET cards, with external mux• 1:N: Yes, with external mux, not supported now Software: <ul style="list-style-type: none">• IOS on VIP cards• Line card IOS "microcode" image bundled with RP IOS and/or resident on IOS file system device

24.7.2.1 Cisco 7500 Forwarding and Control Architecture

The 7500 system has been widely deployed in a wide range in switching and routing applications.

The 7500 supports both centralized and distributed forwarding and includes the following major components:

- **VIP**—Versatile Interface Processor card runs IOS, performs distributed forwarding for those IOS features which support distributed forwarding, and accepts a variety of Port Adapter (PA) daughtercards which can terminate different media types. Multiple VIPs may be installed in a chassis.

The 7500 also supports various types of Interface Processors, which do not include support for distributed switching nor replaceable Port Adapters. Examples of interface processors are the ATM interface processor (or AIP), the serial interface processor (SIP), and the Ethernet interface processor (EIP).

- **RSP**—The Route Switch Processor runs IOS and performs forwarding operations for those features which are not supported on the VIP cards. The RSP includes the system packet memory, known for historical reasons as MEMD, which is shared between all interface cards (VIP or otherwise) and the RSP. The larger 7500 chassis may be configured to contain two RSPs for redundancy. Today, RSP redundancy is per 7500 HSA.
- **CyBus**—The CyBus is a 1- to 2-Gb/s shared bus connecting interface processors with MEMD and the RSP. It supports burst transfers as well as atomic queueing and arithmetic operations on a pool of packet buffers and queue headers.

The 7500 may forward a transit packet in several ways:

- In distributed fashion using a single VIP.

Here, the packet enters one interface attached to a VIP card and leaves by another interface attached to the same VIP. This is known as local switching. In this case the packet is not transferred to MEMD; instead it is buffered locally on the VIP card.

- In distributed fashion using two VIPs.

In this case the packet is transferred from the input switchover VIP card buffer memory to MEMD, and from there dequeued for output by the output VIP card.

- In centralized fashion using the RSP.

Supported features which are not distributed across multiple VIPs (such as MLPPP configured on interfaces terminating on more than one VIP card) or which are not distributed at all are implemented on the RSP. Here packets are again transferred into and out of the shared MEMD as part of switching.

Switchover from the active RSP to the standby RSP involves temporarily suspending transfers over the CyBus and into and out of the shared MEMD. This is known as a “CyBus quiesce.”

Various forms of IPC are used on the 7500. One form runs over the “Dbus” maintenance bus. Another form, using messages known as “love letters,” uses reserved parts of MEMD. The amount of data that may be transferred via a love letter is small, on the order of a few bytes.

24.7.3 Cisco 12000 GSR Overview

The Cisco 12000 GSR family was the first example of the current-generation of Internet core routers and, of those, remains the most successful and widely deployed. Over time the GSR 12000 has been enhanced to support high- and low-speed edge aggregation applications.

The GSR supports distributed, hardware-based forwarding. GSR line cards include microcoded and hardwired forwarding engines which process transit traffic. As with the 7500 VIP cards, the GSR line cards include a local microprocessor which runs an IOS image tailored to the characteristics and operational requirements of the line card.

Unlike with the 7500, the GSR line card IOS image is not normally involved in packet forwarding. The line card IOS image does apply configuration and forwarding control parameters to the line card hardware (including to any specialized forwarding ASICs present) as they are downloaded to the line card by the GSR RP (GRP).

There is one GLC IOS image per GLC. In a GSR 12016 system 10 or more separate IOS instances may exist in the chassis.

Generally, the GRP does not forward transit packets. As a result, the GSR system maintains an effective separation of system control and packet forwarding.

Table 24-5 provides a taxonomy of the architectural elements of the Cisco 12000 GSR:

Table 24-5 Cisco 12000 Gigabit Switch Router

Architectural Element	Description
System	Protocols: IP Separate control & data planes: Mostly, LC CPU does some forwarding? Interconnect: Proprietary, redundant crossbar cell switching fabric
Direct LC-LC I/O	Yes
Standby RP/FP access to interconnect & line cards	No?
Route Processor	Has NME: Yes Redundancy: Yes, per HSA, RPR, RPR+
Forwarding Processor	Multiple, distributed, implemented by custom ASICs + microcode, atomic with LCs Buffering: Distributed input and output on line cards

Table 24-5 Cisco 12000 Gigabit Switch Router (continued)

Architectural Element	Description
Line Cards	<p>Modular media interface (port adapters): No</p> <p>Redundancy</p> <ul style="list-style-type: none"> • 1:1: Yes, SONET APS for SONET cards, with external mux • 1:N: ? <p>Software:</p> <ul style="list-style-type: none"> • IOS on LC control processors • Microcode on HW forwarders • Line card IOS and microcodes images bundled with RP IOS and/or resident on IOS file system device

24.7.3.1 Cisco 12000 GSR Forwarding and Control Architecture

The 12000 GSR was originally designed for the Internet core routing application. More recently the GSR has been developed to support aggregation of higher-speed (DS3 and OC3) interfaces.

The GSR uses distributed forwarding. Transit packets are switched by the input line card, using various forms of hardware assist. The GRP is not involved in packet forwarding.

A crossbar cell switching fabric interconnects the line cards and the GRP. A diagnostic bus (the “Mbus”) is used for low-level system control and IPC.

Today, the GSR supports the RPR redundancy scheme.

IP Services

This chapter includes information on the following IP services available with Cisco IOS:

- IP API Functions
- BEEP (*New in 12.2S RLS7*)
- DHCP
- DNS

25.1 IP API Functions

The following IP API functions are available in Cisco IOS:

- 1 `ip_address_duplicate()`
- 2 `ip_checksum_adjust_long()`
- 3 `ip_checksum_adjust_short()`
- 4 `ip_forus()`
- 5 `ip_ouraddress()`
- 6 `ip_reply_srcadr()`
- 7 `ip_reply_srcadr_check_alias()`
- 8 `ip_reply_srcadr_check_idb()`
- 9 `ip_route()`
- 10 `ip_route_and_transmit()`
- 11 `ip_sendself()`
- 12 `ip_set_proto_outcounter()`
- 13 `ipaddr_to_string()`
- 14 `ipbuild_head()`
- 15 `ipcrc()`
- 16 `ipmwrite()`
- 17 `ipsendnet()`
- 18 `ipttl()`
- 19 `ipwrite()`
- 20 `ipwrite_allow_zero()`
- 21 `ipwrite_prerouted()`

For detailed information on these functions, see the *Cisco IOS API Reference*.

25.2 BEEP

This section introduces the API functions that were added into Cisco IOS to provide support for BEEP (Blocks Extensible Exchange Protocol) core bi-directional transport. BEEP support is based on Open Source version BEEPCORE-C (see also EDCS-355394) and is new in 12.2S RLS7. TCP is the only transport implemented for BEEP in IOS. BEEP is required to support the RFC 3195 Secure Syslog implementation planned for future IOS releases. For more information on BEEP, see:
<http://beepcore.org/>

A BEEP session consists of one or more logical channels. IOS BEEP initially supports a minimum of 16 simultaneous BEEP sessions and 4112 total concurrent BEEP channels.

A BEEP profile includes a unique URI (Uniform Resource Identifier) for each open channel and defines how messages are sent over the channels. There are two types of BEEP profiles, exchange and tuning. Exchange profiles are used to dictate how messages are exchanged with another BEEP peer over a BEEP channel (for example, for handling SOAP (Simple Object Access Protocol) messages) and tuning profiles are used to affect the properties of a whole BEEP session (for example, for encryption or authentication).

For BEEP connections initiated outside of Cisco IOS, it is necessary for a BEEP listener to listen at a TCP port for incoming BEEP connections. By default, there is no IOS BEEP listener, so your application mapping must allocate a TCP port for its own use to listen for BEEP connections.

25.2.1 Terms

BEEP

Blocks Extensible Exchange Protocol

exchange profile

A convention used by a protocol running over a BEEP channel to exchange messages with another BEEP peer. The exchange profile defines the protocol that the application wishes to use.

tuning profile

The profile used by BEEP to determine the session transport type and that will negotiate encryption, if required. The tuning profile defines the basic security of the session.

URI

Uniform Resource Identifier. Type of formatted identifier that encapsulates the name of an Internet object, and labels it with an identification of the name space, thus producing a member of the universal set of names in registered name spaces and of addresses referring to registered protocols or name spaces. [RFC 1630] (Source: Cisco Dictionary of Internetworking Terms and Acronyms)

25.2.2 The BEEPCORE-C API

The BEEPCORE-C API functions are described at:

<http://beepcore-c.sourceforge.net/index2.html>

25.2.3 How to Use the BEEPCORE-C API

Use the BEEPCORE-C API to author profiles for your specific protocol needs, or to use an existing profile. Your application will interface with the profile API and the BEEP IOS wrapper for initiating and terminating connections, or starting listeners. The following subsections provide information on authoring new BEEP profiles and using existing BEEP profiles.

25.2.3.1 Authoring New BEEP Profiles

For information on authoring BEEP profiles, see the following tutorial:

<http://beepcore-c.sourceforge.net/ProfileTutorial.html>

25.2.3.2 Using Existing BEEP Profiles

For an example of how to use an existing BEEP profile, follow the steps involved in using the NULL echo profile:

Step 1 As with all BEEP profiles, you must first call its initialization function (passing a config_obj), and store the resulting PROFILE_REGISTRATION pointer:

```
#include "../??/bp-null_profiles.h"
PROFILE_REGISTRATION *pr;
config_obj *appconfig;

/* create a configuration object */

if (!(appconfig = config_new (NULL))) {
    log_line (LOG_PROF, 6, "config_new: failed\n");
    return 1;
}

if ((pr = null_echo_Init(appconfig)) == NULL) {
    log_line (LOG_PROF, 6, "null_echo_Init: failed\n");
    return 1;
}
```

Step 2 Your next step is to make the connection:

```
DIAGNOSTIC *d;
BP_CONNECTION *bp;

if ((d = tcp_bp_connect (hostName, atoi (portNo), pr, appconfig, &bp))
    != NULL)
    log_line (LOG_PROF, 6, "[%d] %s\n", d -> code, d -> message);
else if ((d = bp_wait_for_greeting (bp)) != NULL)
    log_line (LOG_PROF, 6, "unable to establish session: [%d] %s\n",
              d -> code, d -> message);
```

The BP_CONNECTION and PROFILE_REGISTRATION pointers will be used when requesting a new NULL echo protocol channel.

Step 3 Start a channel using the null_start() function, and store the resulting pointer:

```
void *v;
if (!(v = null_start (bp, pr, NULL)))
    return 1;
```

- Step 4** Send data over the channel using the `null_trip()` function, passing the channel-specific pointer saved from `null_start()`:

```
int cc, len;
char *b1, *b2;

...
/*allocate and prepare outgoing data*/

...
if ((cc = null_trip (v, b1, len, b2, len)) < 0) {
    log_line (LOG_PROF, 6, "null_trip: failed %d\n", cc);
    status = 1;
    break;
}
```

- Step 5** Close the channel using the `null_close()` function, passing the channel-specific pointer saved from `null_start()`:

```
if (null_close (v) < 0) {
    log_line (LOG_PROF, 6, "null_close: failed %d\n", cc);
    status = 1;
}
```

25.3 DHCP

Component developers should become aware of the Cisco IOS DHCP (Dynamic Host Configuration Protocol) implementation specifics described in this section.

25.3.1 DHCP Proxy Client API

The Cisco IOS DHCP proxy client API includes the following functions:

- 1 `dhcpc_get_subnet()`
- 2 `reg_add_dhcpc_get_bootfile_name()`
- 3 `reg_add_dhcpc_get_docsis_security_addr()`
- 4 `reg_add_dhcpc_get_domain_name()`
- 5 `reg_add_dhcpc_get_host_name()`
- 6 `reg_add_dhcpc_get_log_addr()`
- 7 `reg_add_dhcpc_get_merit_dump_file()`
- 8 `reg_add_dhcpc_get_ns_addr()`
- 9 `reg_add_dhcpc_get_server_addr()`
- 10 `reg_add_dhcpc_get_tftp_server()`
- 11 `reg_add_dhcpc_get_time_addr()`
- 12 `reg_add_dhcpc_get_timezone_offset()`
- 13 `reg_invoke_dhcp_client_start_stop()`
- 14 `reg_invoke_dhcpc_free_addr()`

```

15 reg_invoke_dhcpc_free_addr_by_ustring()
16 reg_invoke_dhcpc_get_addr()
17 reg_invoke_dhcpc_lease_address_request()

```

These API functions are described in detail in the *Cisco IOS API Reference*. The following section briefly describes the way these registry functions are used.

25.3.2 Using the DHCP Client Calls

First of all, use `sys/ip/dhcpc.c`, *not* `sys/ip/dhcp_client.c` if you need to reference the source code. Another group created `sys/ip/dhcp_client.c` and `sys/ip/dhcp_client_io.c` for a very specialized function and all of the routines in there are now out-of-date, are not maintained, and will be removed very soon. Do not use them.

DHCP Client has one main routine that is used to get an address. The address can be for a particular interface or just for internal use elsewhere (for example, PPP code requesting a DHCP address so that it can pass the address via IPCP to the client on the other end of the connection).

To use the DHCP client calls, follow these steps:

Step 1 Before calling the routine to get an address, you *must* call:

```
reg_invoke_dhcp_client_start_stop(TRUE, DHCPC_RUN_CLIENT);
```

Note The plan is to eventually make the DHCP Client code “self-starting” such that this call will be no longer needed, but even after the change is made, calling this will not hurt.

Step 2 After the DHCP Client has been started, an address can be obtained by calling the `reg_invoke_dhcpc_get_addr()` function:

```
reg_invoke_dhcpc_get_addr (ipaddrtype *request_addr, idbtype *idb,
                           uchar *ustring, uchar *poolname,
                           ulong wait_ticks, ipaddrtype *mask,
                           boolean proxy, ushort cid_type,
                           ushort cid_len, ulong options)
```

The parameters in `reg_invoke_dhcpc_get_addr()` are defined in the `dhcpc_params` structure. The `wait_ticks` parameter can be zero, in which case the API is non-blocking. The `reg_invoke_dhcpc_get_addr()` call is simply a front-end for the more complex `reg_invoke_dhcpc_lease_address_request()` call, which allows all the caller-specified values:

```
reg_invoke_dhcpc_lease_address_request (ipaddrtype *request_addr,
                                         idbtype *idb,
                                         void *parameters);
```

Where:

`request_addr` is the location where the DHCP address should be written once it has been acquired. If `NULL`, the address will not be written anywhere (this is useful if you want an address to be assigned to an interface and do not need to know the final address assigned).

`idb` is the software IDB of the interface to use when sending the DHCP packets.

parameters is a structure of type `dhcpc_params`:

```
/*
 * Calling parameters for DHCP client.
 */
typedef struct dhcpc_params_ {
...
parameters
...
} dhcpc_params;
```

All the `dhcpc_params` structure calling parameters for DHCP clients are described in the API reference page for the `reg_invoke_dhcpc_lease_address_request()` registry function. This function can be called again later with the same parameters to check on the status of the allocation. All DHCP API and structure definitions are in `sys/ip/dhcp.h`.

- Step 3** Other routines can be called to gather DHCP Option information after the allocation has been made. These routines include:

- `reg_add_dhcpc_get_ns_addr()`—Gets the name-server information from the DHCP server information associated with the specified software IDB.
- `reg_add_dhcpc_get_tftp_server()`—Returns the TFTP server address received in Option 66 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_time_addr()`—Returns the Time Server address received in Option 4 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_log_addr()`—Returns the Log Server address received in Option 7 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_server_addr()`—Returns the DHCP Server address received from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_timezone_offset()`—Returns the TimeZone Offset received in Option 2 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_bootfile_name()`—Returns the Bootfile name received in Option 67 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_docsis_security_addr()`—Returns the Security Server address received in Option 128 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_merit_dump_file()`—Returns the Merit Dump File name received in Option 14 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_host_name()`—Returns the Host Name received in Option 12 from the DHCP server associated with the specified software IDB.
- `reg_add_dhcpc_get_domain_name()`—Returns the Domain Name received in Option 15 from the DHCP server associated with the specified software IDB.

These API functions are described in more detail in the *Cisco IOS API Reference*.

- Step 4** After having acquired a lease on an address, you can call the `reg_invoke_dhcpc_free_addr()` registry function to send a DHCP RELEASE for the address and free up all lease information associated with it:

```
reg_invoke_dhcpc_free_addr(ipaddrtype idb_ip_addr, idbtype *idb);
```

You can also call the `reg_invoke_dhcpc_free_addr_by_ustring()` registry function to free up a lease, by specifying the `ustring` (client-id) and `cid_len` (length of `ustring`) and the software IDB used in the original lease request call:

```
reg_invoke_dhcpc_free_addr_by_ustring (uchar *ustring,
                                         ushort cid_len, idbtype *idb);
```

25.4 DNS

This section describes the functions and data structures that provide Domain Name Service (DNS) services in the Cisco IOS software.

VRF-aware DNS was made available in 12.4T.

25.4.1 DNS API

The following API functions are described in the *Cisco IOS API Reference*:

- 1 `domain_address_lookup()`
- 2 `domain_FQDN_lookup()`
- 3 `domain_free_query()`
- 4 `domain_name_lookup()`
- 5 `domain_quick_addr_lookup()`
- 6 `domain_rectype_lookup()`
- 7 `domain_submit_domain_query()`
- 8 `domain_submit_query()`
- 9 `name_cache_lookup()`
- 10 `name_entry_lock()`
- 11 `name_entry_unlock()`
- 12 `name_lookup()`

One of the most important things to keep in mind when using these routines is that if you call a routine that returns a `nametype` pointer, that pointer is not protected or locked against being cleaned up and freed. This is fine as long as your process does not do anything that results in a context switch. If you think your process might switch contexts before you are done with the `nametype` pointer, you should call the `name_entry_lock()` routine to protect the information, and then call `name_entry_unlock()` when you are done with the `nametype` pointer. Failing to call `name_entry_unlock()` after you are done with the pointer will prevent the system from ever freeing up the memory associated with that entry, even after the entry has completely timed out.

25.4.2 Using the DNS non-blocking API

To submit the DNS query, use:

```
domain_submit_query(hostname, querytype, callback, context);
```

Where *callback* is a function that will be called when you have the answer (or have a definitive “no such hostname” result). For instance, the *callback* function is called as follows:

```
callback(context, resptr);
```

Where *context* is the same *context* you gave in the `domain_submit_query()` function and *resptr* is a pointer to a `dnstype` struct (defined in `servers/domain.h`). This structure contains all of the information about your request and the result. In particular “`resptr->nameptr`” is a pointer to the `nametype` record that contains your information (the same that you would normally get as the returned value from the `domain_rectype_lookup()` call).

The way the `dnstype` structure works is that the `DNSRES_DONE` flag in the `flags` field will be set when the DNS routines are “finished”. You can trust the value contained in the `nameptr` field of the structure *only* if this flag is set. If this flag is set and the `nameptr` field is `NULL`, then no answer was found. This could mean the hostname does not exist or that the name-server(s) were not accessible. If the hostname does not exist, then the `soa` field will point to the SOA record (if any) that was returned along with the negative response. (There should be one for any negative response.) The other flag values defined in the `domain.h` file are really only for internal use.

The `domain_submit_domain_query()` function is used to do nonblocking lookup of a domain name, and it looks in the internal cache first. This function allows the default domain name to be overridden:

```
dnstype *domain_submit_domain_query(char *hostname,
                                     char *domainstring,
                                     int querytype,
                                     dnsresolver_cb callback,
                                     void *context);
```

The `domain_submit_query()` function is the same as the above, but it appends default domain(s) as needed:

```
dnstype * domain_submit_query (char *hostname,
                               int querytype,
                               dnsresolver_cb callback,
                               void *context);
```

P A R T 5

Hardware-Specific Design

Porting Cisco IOS Software to a New Platform

Added new section 26.1.4 “Adding a New CPU Type”. (March 2010)

Added new section 26.1.5.2.1 “Reasons for Stack Overflow”. (April 2009)

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 such that the address of the datum corresponds to the address of the least-significant byte (LSB). This method is called little endian or low-high order. The Intel x86 family of CPUs uses this method. Other CPUs store data such that the address of the datum corresponds to the address of the most-significant byte (MSB). This method is called big endian or the high-low order. The Motorola 680x0 family of CPUs uses this method. Until now, released versions of the Cisco IOS software have run only on big-endian microprocessors (680x0 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 discusses 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.

Note Cisco IOS porting questions can be directed to the ios_platforms@cisco.com alias.

26.1 Portability Issues

This section discusses the following topics related to writing portable code:

- Byte Order
- Data Alignment

- Data Size
- Adding a New CPU Type
- Other Portability Issues

26.1.1 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

26.1.1.1 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 {
        uchar     byte[4];
        ushort   sword[2];
        ulong     lword;
    } d;
} charlong;

charlong cl;
```

You can 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.

Assume that the given byte stream values are hexadecimal (that is, 0x11 0x22 0x33 0x44). Otherwise if they are decimal values, the result is different.

For both little-endian and big-endian:

```
cl.d.byte[0] = 0x11; cl.d.byte[1] = 0x22; cl.d.byte[2] = 0x33; cl.d.byte[3] = 0x44;
```

The difference is in the values of `sword` and `lword`:

1) For little-endian, the values are:

```
cl.d.sword[0]=0x2211; cl.d.sword[1]=0x4433; cl.d.lword=0x44332211
```

2) For big-endian, the values are:

```
cl.d.sword[0]=0x1122; cl.d.sword[1]=0x3344; cl.d.lword=0x11223344
```

Note Unions discourage modularity. Using inheritance/structure layering is generally a better idea.

26.1.1.2 Bit Fields

Bit fields are assigned left to right on some machines and right to left on the others. This means that although bit fields are useful for maintaining internally defined data structures, the question of which end comes first has to be carefully considered when picking apart externally defined data; programs that depend on such things are not portable.

For example, if the bpduhdrtype structure is coded as below, it will have the intended performance only for the big-endian CPUs:

```
typedef struct bpduhdrtype_ {
    ushort protocol;                      /* protocol ID */
    uchar version;                        /* version identifier */
    uchar type;                           /* BPDU type */
    volatile uchar tc_acknowledgement: 1; /* topology change
                                             acknowledgement */

    volatile uchar notusedflags: 6;        /* topology change flag */
    volatile uchar tc: 1;
    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;
```

Note that the ordering of bit-fields within machine words is implementation-defined, and therefore, what would be construed as a reference to `tc-acknowledgement` on big-endian CPUs would be interpreted as a reference to `tc` on little-endian machines. In order for the `bpduhdrtpe` structure to work the same on little-endian CPUs, it must be defined as follows. Note the difference in the order of the bit field.

```
typedef struct bpduhdrtype_ {
    ushort protocol;                      /* protocol ID */
    uchar version;                        /* version identifier */
    uchar type;                           /* BPDU type */
    volatile uchar tc: 1;                 /* topology change flag */
    volatile uchar notusedflags: 6;
    volatile uchar tc_acknowledgement: 1; /* topology change
                                             acknowledgement */

    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_ {
    ushort protocol;                                /* protocol ID */
    uchar version;                                  /* version identifier */
    uchar type;                                    /* BPDU type */
    volatile uchar topology_change;                /* topology change
                                                acknowledgment */

    uchar root_id[1DBYTES];
    uchar root_path_cost[4];
    uchar bridge_id[1DBYTES];
    uchar port_id[2];
    uchar message_age[2];
    uchar max_age[2];
    uchar hello_time[2];
    uchar forward_delay[2];
} bpduhdrtype;

bpduhdrtype bpdu;
```

The following `#define` statements could be defined:

```
#define TC_ACKNOWLEDGEMENT 0x01
#define TC                      0x80
```

The following code, which is nonportable, tests and sets the acknowledgement 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)
    topology_change_acknowledged(span);

bpdu->topology_change = (bpdu->topology_change & ~TC_ACKNOWLEDGEMENT) |
                        (port->topology_change & TC_ACKNOWLEDGEMENT);
```

Note Although bit fields still exist in the code mainly because of the amount of code that would need to otherwise change, you should avoid using them. Masking is a portable approach.

26.1.1.2.1 Structure Padding

Compilers often pad the internals of data structures to optimize and/or avoid alignment errors. This can also be source of portability errors. Message passing and/or memory sharing between differing architectures, such as between line cards and host platforms, can cause problems. Unfortunately, there isn't a one-size-fits-all solution for this, but it helps to be aware of it. For example:

```
struct Me {
    char x;
    short y;
    short z;
};
```

may pad differently depending on the compiler/cpu-architecture. One compiler may pad x out to two bytes and place y in the next 2 contiguous bytes with z padding out to 4 bytes; another may pad x out to four bytes and place y and z next to each other; and still another compiler may pad everything out to 4 bytes. This is just for 32-bit architectures; 64-bit and 16-bit CPUs may pad entirely differently. Note that the padding situation tends to get worse when dealing with bit fields.

26.1.1.2.2 Unary Types

Consider using unary types with bit-position enumerations instead of bit fields as it would also help for structure padding. When writing unary types, use intrinsic types like short or long. For example:

```
struct You {
    ushort flag;
    ...
};

enum YouFlag {
    YouFlag_enabled = 0x0001,
    YouFlag_something = 0x0002,
    ...
};
```

This way, you are writing code that ensures that endianness is not a problem.

```
struct You you;
...
if (you.flag & YouFlag_enabled)
    ...
```

26.1.1.3 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. Caution should be exercised when using the right shift operator (>>). Right shifting a signed quantity will fill with sign bits on some machines and with 0-bits on others.

26.1.1.4 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;
```

Similarly, the following code, seen commonly while reading packet data, gives different results on CPUs with different byte orders when an attempt is made to dereference either s, i, or l.

```
char a[20];
short *s=&a[0];
int *i=&a[0];
long *l=&a[0];
```

26.1.2 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 680x0 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.

26.1.2.1 MC68000 Memory Addressing Examples

Programming model: hardware: word addressable (16 bits for MC68000)

Address (hex)	Data (hex)
000000	3c80
000002	852f
000004	82cc
...	
fffffe	fa37

To read a byte stored at \$000001:

```
read word @ $000000: $3c80
CPU extracts second byte: $80
```

Word Alignment

```
* read word @ $000002
one memory access

* read word @ $000001
two memory access
disallowed, causes a memory trap
```

To read a long word stored at \$000002:

```
involves two memory accesses, each fetching 2 bytes
invisible to programmer
final result: $852f82cc
```

In summary:

- words and long words must be word aligned
- even memory addresses

26.1.3 Data Size

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 cases for assuming the type `int` instead of a predefined type like `uint_t`:

- 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.

26.1.4 Adding a New CPU Type

Every time you port Cisco IOS to a new CPU variant, you need to add a new CPU type. The process involved in adding a new CPU type is to either reuse what is existing to the extent possible, or code the new .h file and send the diffs out to:

- 1) codereviewer-os and
- 2) codereviewer-cpu

Specific guidelines for what needs to be done are not available, but several new header files must be defined. By convention there is a header file for the CPU in:

- `machine/cpu_<cpu>.h`,
- `machine/cisco_<cpu>.h`
- a platform-specific header file named `machine/cisco_<platform>.h`

The CPU header file has definitions for the CPU and compiler (gcc) combination such as:

- Endianess
- Defines or inlines for `GETSHORT/PUTSHORT/GETLONG/PUTLONG` and the like for writing to misaligned memory (You must check your processor's alignment rules)
- Fundamental typedefs (`u_char`, `u_short` and the like)
- Number of registers
- Register structure and perhaps names
- Processor status word
- `current_stack_pointer` function, which returns its namesake
- Atomic increment and decrement routines using the appropriate assembly instructions for the processor
- Interrupt enable and disable routines
- Semaphore lock/unlock routines
- Defines for the size of certain types (`LONG/SHORT`) when they are pushed on the stack (probably for function calls)
- CPU-specific trap and vector definitions
- Other, CPU-specific definitions, for example `#defines` and inlines for dealing with a particular CPU's cache.

Note You should try and reuse the existing PPC headers wherever appropriate (PPC is a power PC, a processor family type like MIPS).

The machine CPU file is really a platform-specific file. It defines platform-interrupt assignments and levels for specific devices that should always be in a Cisco IOS box, for example a Console and Ethernet port. It defines the number of lines (TTYs) on the system, as well as system clock timer parameters. Among other things, it includes `image_<file format>.h`, which contains download file format-specific defines that must match the build tool chain. All these various header files are selected by conditional compilation in `target_cisco.h`.

26.1.5 Other Portability Issues

The following issues also have an impact on writing portable code:

- Performance
- Stack Usage and Stack Growth
- Compliance with Encapsulations

26.1.5.1 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.

For example, consider the function `enqueue_inline()`, which adds an element to a fifo queue. Assume that `qptr` is a pointer to a global data structure. This piece of code works perfectly well in an IOS-like nonpreemptive environment.

```
static inline void enqueue_inline (queuetype *qptr, void *eaddr)
{
    nexthelper *p, *ptr;
    p = qptr->qtail;           /* last element pointer */
    ptr = eaddr

    if (sanity_debug && !is_valid_ram((void *)ptr)) {
        errmsg(&msgsym(LINKED, SYS), queue_badEnqueueStr, ptr, qptr);
        crashdump(0);
    }

    /*
     * Make sure the element isn't already queued or the last element
     * element in this list
     */
}
```

```

if ((ptr->next != NULL) || (p == ptr)) {
    (*kernel_errmsg)(&msgsym(LINKED, SYS), queue_enqueueStr, ptr, qptr);
    return;
}
if (!p)                      /* q empty */
    qptr->qhead = ptr;
else                         /* not empty */
    p->next = ptr;           /* update link */
qptr->qtail = ptr;          /* tail points to new element */
qptr->count++;
}

```

Note If this function was called within a Windows NT-like preemptive environment, this code would not work. Because there is no guard when `qptr` is being modified, a context switch can occur half way through the modification causing the new running process to perhaps try to alter `qptr`, eventually leaving it in an inconsistent state when the original process resumes execution.

26.1.5.2 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. 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 push the argument from left to right on the stack.
- The code assumes that all arguments 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`.

26.1.5.2.1 Reasons for Stack Overflow

The following are possible causes for stack overflow:

- Using large *local* array or structure variables in a function.
- Infinite or excessively deep recursive function calls.
- Excessively deep function call sequences (large number of functions in a particular function call sequence up to the leaf function).

Note Be aware of the consequences of using `#defines`. This can cause small arrays to become too large. When using arrays as local variables in the stack, we need to evaluate whether the array could grow in the future and, if yes, it is better to allocate it (preferred) or have a comment before `#define` saying that increasing the array length could cause stack overflow. This applies to structures as well.

26.1.5.3 Register Considerations

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 (char, unsigned char, short, unsigned short, long, and unsigned long), but register a0 might be used for returning addresses (char *, void *, short *, and so on).

One place where it is possible to encounter this type of problem is when calling an assembly language function from C language code, or when calling a C function from assembly language code. Care must be taken to ensure that the returning routine places the return value in the correct register.

26.1.5.4 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.

Consider the task of writing the HDLC header into the paktype structure:

```
#define HDLC_BRIDGECODE 0x0F006558L      /* HyBridge encapsulation code */
#define HDLC_STATION    0x0F00                /* station address for raw HDLC */
#define TYPE_BRIDGE     0x6558                /* serial line bridging */

typedef struct_vhdlctype {
    ushort var_hdlcflags;
    ushort var_hdlctype;
    uchar data[0];
} vhdlctype;
hdlc = (vhdlctype *) pak->datagramstart);
```

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 using the PUT macros. The PUT macros, as will be explained later, take as input the address where the short or long value should be placed, and also the value that should be placed at that address.

```
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;
    uchar sde_said[4];                                /* Security association ID */
    uchar sde_sid[8];                                 /* Station ID */
    uchar sde_flags;                                  /* Flags */
    uchar sde_fid[4];                                /* Fragment ID */
} sdehdrtype;

/*
 *This is the 802.10 Security Association data structure. It holds the SAID
 'color' field of the security association and the sub-interfaces configured
 with that SAID. The SAID values are expected to be unique throughout the
 network. We maintain an array, indexed by hardware interface number, of the
 software IDBs associated with this SAID value, so that when we match a
 received 802.10 packet's SAID with those configured on the received
 interface, we can then index the corresponding sub-interface IDB -- which
 ultimately allows us to get the packet to the correct bridge group. Note
 that there can only be one sub-interface per physical interface with a given
 SAID, but multiple sub-interfaces (of different hardware IDBs) can be
 configured with the same SAID. For redundancy, let's keep this longword
 aligned...
*/
struct said_descriptor_ {
    ulong    sdb_said;                                /* Security Association ID */
    uchar    sdb_fragmentation;                         /* Do we exceed the MTU? */
    uchar    sdb_encryption;                            /* Packet data encrypted? */
    uchar    sdb_integrity_check;                      /* Has packet been modified? */
    uchar    sdb_number_subif_in_bgroup;                /* Statistical count */
    said_descriptor *next_said_db;                     /* List of descriptors */
    ulong    ingressing_dot10_packets[VLAN_PROTOCOLS_MAX];
    ulong    egressing_dot10_packets[VLAN_PROTOCOLS_MAX];
    encapstype interior_encapsulation;                /* Novell IPX only */
    /*
     * This is a "zero-length" array, which is a GCC extension to ANSI C. No
     * storage for these arrays is allocated in the structure; we will provide
     * the appropriate allocation when structures of this type are malloc'd.
     * This element MUST be last in the struct.
     */
    /* Software IDBs associated with this SDE entity */
    idbtype *sdb_sub_interface[0];                     /* will be MAX_INTERFACES */
define SAP_SDE           0x0A                      /* 802.10 Secure Data Exchange */
define LLC1_UI            0x03
```

The following code writes the header to a buffer. The GETmacros take as input the address where the short or long value should be read and they return the requested short or long value.

```
/*
 * Write an 802.10 SDE header to an network buffer.
 *
 */
void dot10_header_wr_bfr (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);
}
```

As seen above, the code takes care of the alignment issues by making calls to PUT and GET macros. But seen from the endian perspective it disregards how values are read and written.

26.2 Cisco's Implementation of Portability

This section discusses some features of the Cisco IOS code that enable the writing of portable code.

26.2.1 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.

26.2.2 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

26.2.3 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.

26.2.4 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 call 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 endian type should be assumed. 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
```

26.2.5 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)
```

26.2.6 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 macro `BYTE_ORDER`. 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_BYTESHORT(&ip->t1);
    ORDER_BYTESHORT(&ip->id);
    ORDER_BYTELONG(&ip->srcadr);
    ORDER_BYTELONG(&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_BYTELONG(&ptr->hops[j]);
                    break;
                }
                case IPOPT_TSTMP:
                {
                    ipopt_tstmptype *ptr = (ipopt_tstmptype *)&opts[i];
                    int j, ntimes = btol(ptr->length - IPOPT_TSTMPHEADERSIZE);
                    if ((ntimes < 1) || (ntimes > btol(MAXIPOPTIONBYTES)))
                        return;
                    for (j = 0; j < ntimes; j++)
                        ORDER_BYTELONG(&ptr->tstmp[j]);
                    break;
                }
            }
        }
    }
}
```

```
    if ((ntimes < 0) || (ntimes > btol(MAXIPHEADERBYTES)))
        return;
    for (j = 0; j < ntimes; j++)
        ORDER_BYT_LONG(&ptr->tsdata[j]);
    break;
}
case IPOPT_SID:
{
    ipopt_sidtype *ptr = (ipopt_sidtype *)&opts[i];
    if (ptr->length != IPOPT_SIDSIZE)
        return;
    ORDER_BYT_SHORT(ptr->streamid);
    break;
}
}
}
```


P A R T 6

Management Services

Command-Line Parser

Updated section 27.7.3 “Important Notes on Data Variables”. (October 2010)

Added a note about the impact of white spaces within keywords. (October 2010)

Added a new section 27.3.7.3 “Difference between NONE and no_alt”. (August 2010)

Added a note in the section 27.13.1 “Add a Parser Mode”. (July 2010)

Removed all parser dump utility commands from Table 27-7 “Useful show parser Commands” because these commands were deprecated. (July 2010)

Added PRC error codes in the section 27.5.2.1 “Error Codes”. (June 2010)

Removed the section 27.3.7 “Usage of ALTERNATE, no_alt and NONE” (June 2010)

Added the newly created OPTIONAL_KEYWORD macro to Table 27-1 “Macros for Parsing Keywords”. (March 2010)

Added new section 27.3.10 “Generating Error Messages in Configuration Mode”. (March 2010)

Added new section 27.12.6 “Limiting Parser Command Searches in Interface Submodes”. (February 2010)

Added additional information to section 27.8 “Ordering Commands”. (October 2009).

Added new section 27.3.2.2 “How to Write Commands for Subinterfaces”. (June 2009)

27.1 Parser Information Sources

This chapter includes a wealth of information that describes the functioning and usage guidelines for the command-line parser. It is intended for an audience who want to implement IOS commands by constructing parse chains that invoke parser macros.

In addition, the IOS CLI website is available:

<http://wwwin-eng.cisco.com/Eng/IOS/CLI/WWW/>

It provides the following information areas:

- IOS CLI development, including the Parser Police Manifesto.
- IOS CLI debugging, including various step-by-step directions for troubleshooting broken parse trees and chains.
- IOS Parser Group, including information that is specific to developers who are responsible for maintaining the parser component.
- Contact information for the parser development group.

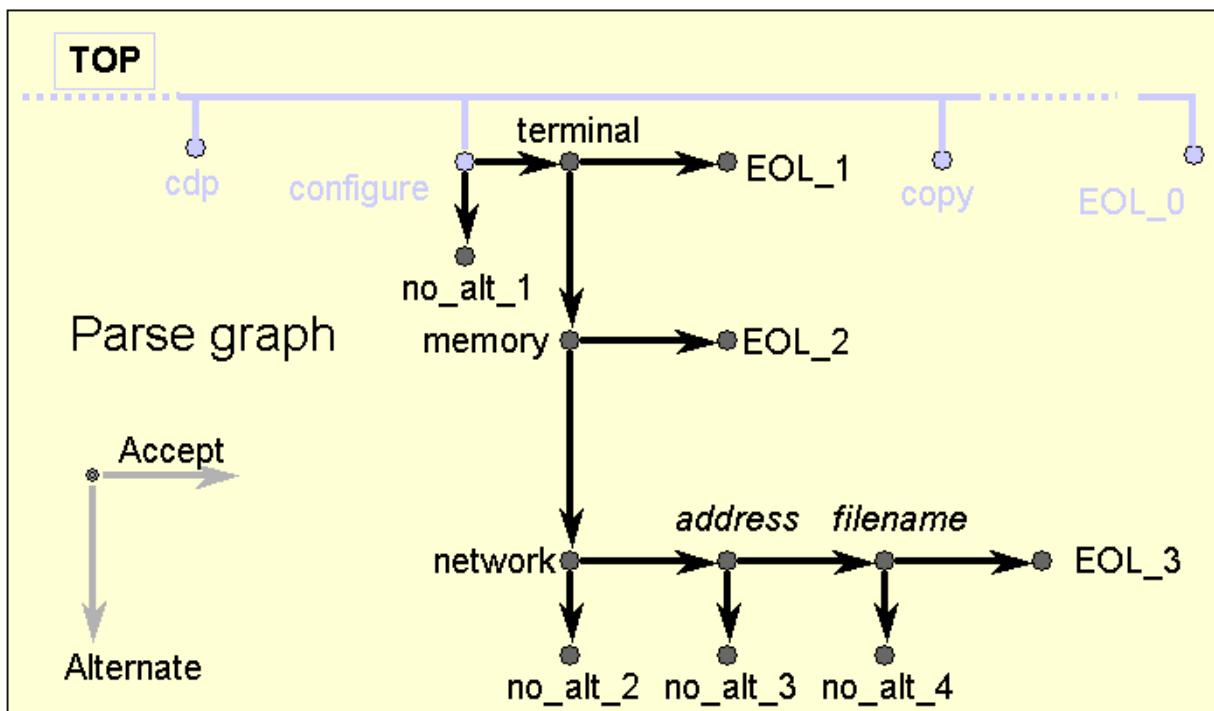
27.1.1 Using CLI Editing Features and Shortcuts

The parser key bindings are documented in the *Cisco IOS Configuration Fundamentals Guide*, available online at www.cisco.com. As the URL is frequently updated, by performing a search for “IOS CLI” on the website, you will find the section that describes the editing features, including tables of key bindings.

27.2 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. To parse the command string, the parser matches token from the command string against nodes in the parse graph. The state of the parser is maintained in a Console Status Block (CSB). See Figure 27-1, “Parser Model” for an example of a parser model showing nodes in a parse graph.

Figure 27-1 Parser Model



Node

A *node* consists of an item to match (usually text), and an *accept* transition and an *alternate* transition. Each *transition* points to another node, or sometimes a function to execute. The node pointed to by the accept transition is called the *accept node*. The node pointed to by the alternate transition is called the *alternate node*.

Nodes are defined by various CLI *macros*. Different macros define different types of nodes, indicating different things to be matched (e.g. text, IP addresses, MAC addresses) and different things to do when a match is made (e.g. go to another node, execute a function, jump to a new parse graph, cause a parser error).

Nodes are also referred to as *parser nodes* or *parse graph nodes*. For more information about nodes, see section 27.3.1, “Construction of Parse Trees” and section , “.”

Parse graph

The *parse graph* is the set of parser nodes. The parser has one main graph and several subgraphs linked onto the main graph and each other. Remember that each parser mode has its own parse graph.

This chapter includes the following topics:

- Building Parse Trees
- Adding Parser Return Codes
- Hot ICE (*New in 12.5pi1, 12.4T, 12.2S*)
- Linking Parse Trees
- Manipulating CSB Objects
- Ordering Commands
- Adding Commands Dynamically
- Manipulating Parser Modes
- Debugging Parser Ambiguity
- Adding New Interface and Controller Types (*New in 12.3T*)
- ICD (Intelligent Config Diff) (*New in 12.3T*)
- IOS Config Rollback (*New in 12.3T*)
- How to Find a Command’s EOL
- NVGEN Enhancements
- Warning Against an Interactive CLI

Note Several common macros shown in this chapter (such as EOLS, EOLNS, PARAMS, and NVGENS), have been replaced by new `xxx_COMMAND` macros in 12.4Tp16 as shown in section 27.5, “Hot ICE”, but remain as-is in other sections of this chapter to represent the generally released code that has not yet been updated.

Note Parser code development questions can be directed to the `parser-questions@cisco.com` alias. New CLI or changes to existing command syntax must be reviewed by `parser-police@cisco.com`. Questions on the formatting of output for new commands and changes to existing commands, including output formatting for debugs and event logs, can be directed to `cli-output-police@cisco.com`.

For questions regarding how the parser works in different branches, contact `parser-dev@cisco.com`. For additional guidelines about developing CLI syntax for new commands, see also:
<http://wwwin-eng.cisco.com/Eng/Process/Release/Parser-Police.txt>

27.2.1 Traversing the Parse Tree

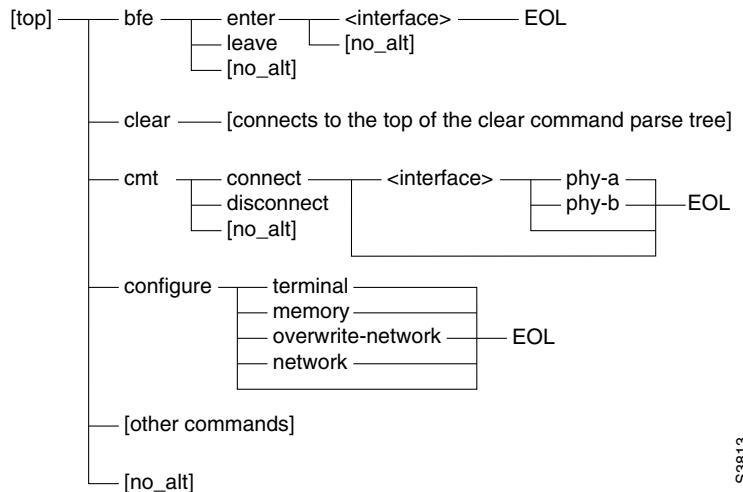
When parsing a command, the command-line parser checks the entire parse tree, searching all alternates for a given token, in order to detect ambiguous input that might occur when abbreviated keywords are used. Although this might seem to present a significant load to the CPU, very few branches of the parse tree are actually traversed for any given command input.

Note White space is not valid within a single keyword. White spaces within keywords break the help functionality in parser. However, there will be no any visible change because of splitting a single keyword into two keywords.

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.

Figure 27-2 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 After reaching the EOL (End of Line), the parser allocates a new console status block (CSB) and copies the current parse data in the Exec CSB to the new 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 27-2 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).

Note The `<cr>` option is displayed by the parser when the EOL node is reached. There is no provision to add help information to the `<cr>` option.

- 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.
- 5 Alternates of **configure** are also parsed till NONE is reached to identify if any other match is there for **configure terminal** command.

27.2.1.1 Parsing Config versus Commands

When the “resolvemethod” from the `csb_struct` is set to `RES_NONVOL`, the parser is parsing a configuration during boot time; otherwise, the parser is parsing actual commands being entered by the user.

The `resolvemethod` flags are for resolving protocol addresses, configuration information, and more, for example:

`RES_BOOTP` is for input from the Boot Protocol.

`RES_CONFIG` is for input from a configuration file.

`RES_DHCP` is for input resolved via DHCP.

`RES_HEURISTIC` is for Heuristics (usually 3MB PUP).

`RES_IPCP` is for input that is negotiated over a PPP/IPCP link.

`RES_MANUAL` is for input from the CLI.

`RES_NONVOL` is for input from the non-volatile RAM.

`RES_NULL` is for input that does not need to be resolved.

`RES_PRIVATE_NONVOL` is for input read from private non-volatile RAM.

`RES_RARP` is for input from reverse ARP.

`RES_WRITING_NONVOL` is for writing to non-volatile RAM.

27.2.2 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;
    struct transition_ *up pointer;
};
```

`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.

`*up pointer` is a pointer to the parent of the node of transition and it is used by the parser internally for the parser chain cache.

Note The help string argument format is a string that can contain white space, does not end with a period, and is delimited by quotation marks. For example, "Turn off privileged commands" or "Enable level". Avoid contractions and be concise in help strings for readability. The longest command, plus six spaces (two spaces before and four spaces after), plus the string should hold to 80 characters or less when on the screen.

27.3 Building Parse Trees

You build parse trees for a command using a set of macros and flags to describe each token in the command. Each macro defines a unique node in a command's state diagram. See the *Cisco IOS API Reference* for details on the macros and flags (such as the `CON_* flags`, `KEYWORD_* flags`, `NUMBER_* flags` and `*_NUMBER flags`).

27.3.1 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.

27.3.1.1 Example: Construction of Parse Trees

The following example of constructing a parse tree shows the parse tree for the **disable** EXEC command:

```
/*
 * disable
 */

EOLS      (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 27-2 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.

27.3.1.2 Parser Chains

There is a standard file naming convention for parser related files. Parser chains should be defined in the following standard files (where `cfg_foo.h` is for commands in config mode, `cfg_int_foo.h` is for commands in interface config submode, and `exec_xxx_foo.h` is for commands in exec mode):

- 1 `cfg_foo.h`
- 2 `cfg_int_foo.h`
- 3 `exec_clear_foo.h`
- 4 `exec_debug_foo.h`
- 5 `exec_show_foo.h`
- 6 `exec_test_foo.h`

7 exec_debug_foo.h

These files should be included in `foo_chain.c`.

27.3.1.3 Maintaining Backward Compatibility

IOS developers have begun the effort to produce a configuration that is compatible with older IOS images. When a newer configuration is submitted to an older IOS image, some functionality can be lost wherever a new syntax goes unrecognized by an older image. The CLI deprecation process, introduced in 12.4T through CSCek50848, provides the internal support needed to produce a downgraded configuration, at the customer's discretion. The downgraded configuration will be compatible with any older IOS image in the same release series, reaching backwards up to 18 months.

Deprecation Process Step-by-Step Procedure

Use the following steps for downgrading a configuration to produce an IOS image that will be compatible with any older IOS image in the same release series:

Step 1 Hide the deprecated syntax. Use `PRIV_HIDDEN` at the appropriate keyword node in the parse chain.

Step 2 Insert a `DEPRECATED_CLI` macro on the accepting transition, just after the newly hidden node. This macro must contain a specific date stamp string parameter; the date itself should be the `show version` build date of the last released image for which this syntax was still visible (not yet hidden). In other words, it's the build date of the IOS release just prior to the current development effort. Date format is YYYY/MM/DD.

Step 3 Do not remove the calls to `nv_write()`, `nv_add()`, `nv_write_default()`, or `nv_add_default()` in their respective action function. Instead, place them inside a code block(s) headed by the following conditional:

```
if (csb->nvgen_deprecated) {
    /* place deprecated calls to nv_*( ) functions here */
}
```

Note This output would be visible during a `show run deprecated`, but not during a `show run`.

Step 4 Create the new syntax via parse chain additions.

Step 5 Create new `nv_*()` calls as needed for the new syntax. Place them inside a code block headed by the following conditional:

```
if (!csb->nvgen_deprecated) {
    /* place replacement calls to nv_*( ) functions here */
}
```

Note This output would be visible during a `show run` (Step 5), but not during a `show run deprecated` (Step 3). Output from (Step 3) and (Step 5) are mutually exclusive. To produce both simultaneously under any conditions would be a bug.

Deprecation Process After 18 Months

Use the following steps to remove the downgraded configuration after a period of 18 months:

- Step 1** Remove the deprecated syntax from the parse chains, including the DEPRECATED_CLI macro.
- Step 2** Remove the csb->nvgen_deprecated code block(s) from the action function.
- Step 3** Remove the !csb->nvgen_deprecated conditional from the action function, but keep the code block contents.
- Step 4** Done! You have now successfully removed the syntax from the deprecation process.

For more information about the deprecation process, see ENG-31399, “Parser-Police Manifesto” at: <http://wwwin-eng.cisco.com/Eng/Process/Release/Parser-Police.txt>.

27.3.1.4 How to Avoid Collateral OBJ Damage in the Parse Trees

Consider this stylized view of the “show” branch of the exec mode parse tree:

```

"show" - "mpls" - <"show mpls" accepting path>...
|
(
"show" alternate path)
|
"ssm" - <"show ssm" accepting path>...
|
(
"show" alternate path)
|
"vrf" - <"show vrf" accepting path>...
|
(
"show" alternate path)
|
"tcam" - <"show tcam" accepting path>...
|
(
"show" alternate path)
|
.
.
.

```

An OBJ(type, N) variable, once set, persists for the duration of a parse (this is required functionality). In the example above, OBJ(type, N) can be used in the "mpls" <accepting path>, without concern for how OBJ(type, N) is used in any other chain's <accepting path>, because the act of parsing ensures that only one <accepting path> will be traversed.

Note Hidden chains and duplicate keywords always run the risk of OBJ collisions with ambiguous visible chains, so special attention must be paid to those.

However, if one sets one or more OBJ(type, N) variables from anywhere along the "show" <alternate path>, those non-zero OBJs can have an impact on "show" alternates that follow.

For example, the "show" alternate path might set OBJs 21 and 22 just prior to parsing "vrf", as was done here:

```

/*
 * Parse chain for VPN show command.
 */
#define ALTERNATE NONE
#include "../iprouting/exec_show_vpnh.h"

```

```

LINK_POINT(vrf_ip_show_cmd, ALTERNATE);
SET(vrf_show_set_version, ALTERNATE, OBJ(int, 21), VRF_CLI_VERSION_NEW);
SET(vrf_show_set_afi, vrf_show_set_version, OBJ(int, 22), AF_LIMIT);
LINK_POINT(vrf_show_cmd, vrf_show_set_afi);
#undef ALTERNATE

```

"show vrf ..." will work as expected. Subsequent "show" alternates will be impacted by the fact that two non-zero OBJS are now present, where they used to always be zero. In this particular case, the "vrf" chain additions hardwired a couple "show team" commands to always show the "detail" option without regard to the actual command.

An OBJ's scope can be limited by clearing the values when exiting the relevant area; in this example, one can set both OBJS to zero explicitly when leaving "vrf" along the alternate path. Another option would be to rearrange the chains to avoid using OBJS on the "show" alternate path.

In general, be careful with the scope of csb OBJS and keep the above in mind when allocating them in your chains.

27.3.2 Parse a Keyword Token

To parse a keyword token, use the GENERAL_KEYWORD and KEYWORD_* macros. Table 27-1 lists some common macros for parsing keywords.

Table 27-1 Macros for Parsing Keywords

Parsing Task	Macro
Parse a keyword.	GENERAL_KEYWORD(<i>name, accept, alternate, keyword, help, privilege, variable, value, match, flags</i>)
Parse a keyword and evaluate a C expression.	OPTIONAL_KEYWORD(<i>name, accept, alternate, expression, string, help, privilege</i>);
Parse a keyword, accepting partial matches and ignoring case.	KEYWORD(<i>name, accept, alternate, keyword, help, privilege</i>)
Parse a keyword and set the specified CSB member to the supplied unsigned integer identifier.	KEYWORD_ID(<i>name, accept, alternate, variable, value, string, help, privilege</i>)
Parse a keyword, matching a minimum number of characters.	KEYWORD_MM(<i>name, accept, alternate, keyword, help, privilege, count</i>)
Parse a keyword without parsing the trailing white space.	KEYWORD_NOWS(<i>name, accept, alternate, keyword, help, privilege</i>)
Parse a keyword and optionally parse the trailing white space.	KEYWORD_OPTWS(<i>name, accept, alternate, keyword, help, privilege</i>)
Parse a keyword that is followed by an existing interface.	INTERFACE_KEYWORD(<i>name, accept, alternate, variable, valid_ifs, keyword, help</i>)

The following parameters are common to most of the macros listed in Table 27-1:

- *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 27-2 lists the flags for specifying the keyword privilege level. Specify only one privilege level for a keyword. You can modify them with the **privilege** configuration command. (See also “Understanding Privilege Levels” in Tech Notes Document ID: 23383 at <http://www.cisco.com/warp/public/63/showrun.shtml#priv>.)

Table 27-3 lists the flags for specifying how the keyword should be parsed and NV generated, and how it should provide help. Several values can be ORED together for the *privilege* argument.

Table 27-2 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_NULL	Set to privilege level 0 (the lowest privilege level). Similar to PRIV_MIN.
PRIV_USER	Set the privilege level needed to parse the keyword to 1. This is the default user privilege level.
PRIV_OPR	Set to privilege level 15. Mostly used for EXEC mode commands.
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_ROOT.
PRIV_CONF	Set the privilege level needed to do configuration. Indicates that the CLI is in CONFIG mode.

Table 27-3 Flags for Specifying Other Options When Parsing Keywords

Flag	Description
PRIV_DISTILLED	When a CLI is tagged with the PRIV_DISTILLED flag and a boot configuration command such as boot config slot0:xxx exists, the NVGEN output of that CLI is not written to NVRAM but to the configuration file in flash. For example, when the command boot config slot0:xxx is given, IP access-lists configurations are not written (distilled) to NVRAM
PRIV_INTERNAL	The command is an internal command and is unavailable unless you enter the service internal configuration command.
PRIV_INTERACTIVE	Used to flag an interactive command. When this flag is set for a command, the command is not available for an HTTP-based configuration session.

Table 27-3 Flags for Specifying Other Options When Parsing Keywords (continued)

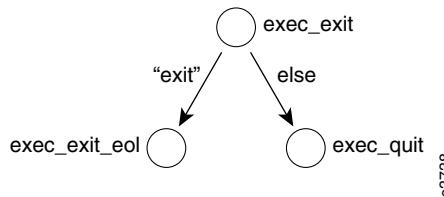
Flag	Description
PRIV_UNSUPPORTED	The command is unsupported and hidden, and no help is provided. If this command is entered, a warning message is displayed indicating that the command is unsupported; the command is then executed.
PRIV_USER_HIDDEN	The keyword is hidden to users with a privilege level 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. To find all hidden commands, grep for the HIDDEN flag in the parser's tree top node macro.
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.
DEFAULT_PRIV	Default user privilege level (level 1). Same as PRIV_USER.
PRIV_NO_SYNALC	The keyword for the Cat5k ATM (SYN for synergy, which was the code name for the Cat5k, and ALC for ATM Line Card). This flag is only used to block commands in Cisco IOS that are not supported by the Cat5k because this method for enabling/disabling commands will not scale for use with other platforms.
PRIV_CHANGE_DISALLOWED	Setting this flag prevents changing the privilege/view information on the keyword. Typically, it is used for security purposes where lowering the privilege may cause security holes.

27.3.2.1 Example: Parse a Keyword Token

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);
```

Figure 27-3 illustrates the transition diagram for this example, showing the three nodes `exec_exit`, `exec_exit_eol`, and `exec_quit`.

Figure 27-3 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
};
  
```

27.3.2.2 How to Write Commands for Subinterfaces

Keywords can be available on subinterfaces. If the keyword does not have a `PRIV_SUBIF` flag, the keyword is unavailable when configuring subinterfaces. Use `PRIV_SUBIF` along with `PRIV_CONF` to enable subinterface configuration.

27.3.3 Parse a Number Token

To parse a number token, use the `GENERAL_NUMBER`, `NUMBER`, and other related macros. Table 27-4 lists some macros for parsing number tokens. See [Chapter 27, “Command-Line Parser”](#), in the *Cisco IOS API Reference Guide*, for usage details and to find additional variations of number-parsing macros.

Table 27-4 Macros for Parsing Numbers

Parsing Task	Macro
Parse a number in a specified range. The <i>flags</i> parameter indicates the numeric criteria against which to match the input number.	<code>GENERAL_NUMBER(name, accept, alternate, variable, lower, upper, help, flags)</code>
Parse a decimal or a hexadecimal number in a specified range. (<i>Octal support deprecated in 12.2S, 12.4, 12.4T; use GENERAL_NUMBER macro with octal-processing flags or OCTAL macro as described in Section 27.3.3.1, “Octal Number Parsing in NUMBER and Other Related Macros”.</i>)	<code>NUMBER(name, accept, alternate, variable, lower, upper, help)</code>
Parse a decimal or a hexadecimal number in a specified range without parsing trailing white space and without providing help. (<i>Octal support deprecated in 12.2S, 12.4, 12.4T; use GENERAL_NUMBER macro with octal-processing flags or IOCTAL macro as described in Section 27.3.3.1, “Octal Number Parsing in NUMBER and Other Related Macros”</i>)	<code>INUMBER(name, accept, alternate, variable, lower, upper)</code>

Table 27-4 Macros for Parsing Numbers (continued)

Parsing Task	Macro
Parse a decimal number in a specified range.	<code>DECIMAL(name, accept, alternate, variable, lower, upper, help)</code>
Parse a decimal number in a specified range without parsing trailing white space and without providing help.	<code>IDECIMAL(name, accept, alternate, variable, lower, upper)</code>
Parse an octal number in the range 0 through 0xFFFFFFFF.	<code>OCTAL(name, accept, alternate, variable, help)</code>
Parse an octal number in the range 0 through 0xFFFFFFFF without parsing trailing white space and without providing help.	<code>IOCTAL(name, accept, alternate, variable)</code>
Parse a hexadecimal number in the range 0 through 0xFFFFFFFF.	<code>HEXNUM(name, accept, alternate, variable, lower, upper, help)</code>
Parse a hexadecimal number in a specified range.	<code>HEXDIGIT(name, accept, alternate, variable, lower, upper, help)</code>
Parse a hexadecimal number in the range 0 through 0xFFFFFFFF without parsing trailing white space and without providing help.	<code>HEXADECIMAL(name, accept, alternate, variable)</code>
Parse a number in the range 0 through 127, or parse a single character. The <i>flags</i> parameter indicates the numeric criteria against which to match the input number.	<code>GENERAL_CHAR_NUMBER(name, accept, alternate, variable, help, flags)</code>
Parse a number in the range 0 through 127, or parse a single character. (<i>Octal support deprecated in 12.2S, 12.4, 12.4T.</i>)	<code>CHAR_NUMBER(name, accept, alternate, variable, help)</code>

The following parameters are specified for most of the macros listed in Table 27-4:

- *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.

27.3.3.1 Octal Number Parsing in NUMBER and Other Related Macros

(*In releases prior to 12.2S, 12.4, 12.4T, and earlier branches of these trains*) The NUMBER and related macros allow input number strings to be parsed with non-strict matching of hexadecimal, octal, and decimal numbers that might cause unexpected results. This behavior applies to the NUMBER and related macros such as NUMBER_FUNC, NUMBER_HELP_FUNC, NUMBER_NOWS, NUMBER_FUNC_NOWS, NUMBER_NV, INUMBER, CHAR_NUMBER, and CHAR8_NUMBER, as follows:

- A number string beginning with 0x or containing the hexadecimal digits A through F is parsed as a hexadecimal number.
- A number string beginning with 0 and containing only the digits 0 through 7 is parsed as an octal number.
- All other number strings are parsed as decimal numbers, including leading-zero numbers that contain non-valid octal digits.

The fall-through number checking to match with hex, octal, or decimal numbers can result in ambiguous interpretation of octal and decimal numbers. For example, in a CLI command that accepts a series of decimal number strings such as (08, 09, 010, 011), using the NUMBER and related macros, the parser would interpret the first two numbers as decimal values 8 and 9 because although they have a leading 0, they contain non-valid octal digits. The parser would then interpret the next two numbers as octal, because each has a leading 0 and contains all valid octal digits, resulting in decimal values 8 and 9. The input most likely was intended to be parsed as decimal values (8, 9, 10, 11), rather than (8, 9, 8, 9).

(*Added with CSCei92592, then later deprecated in 12.2S, 12.4, and 12.4T with CSCsc84077; might still exist in some branches off of these trains*) To avoid possible ambiguity in parsing numbers, you can use the GENERAL_NUMBER macro instead of the NUMBER macro or its variations, with the NUMBER_STRICT flag in the *flags* parameter to enforce strict interpretation of number bases as follows:

- A number string that begins with 0x is parsed as a hexadecimal number only.
- A number string that begins with 0 is parsed as an octal number only. The input is considered invalid if the string contains a non-octal digit, and the parse fails.
- A number that does not begin with 0 or 0x is parsed as a decimal number. The input is invalid if the string contains a non-valid decimal digit, and the parse fails.

In this case, the series of numbers (08, 09, 010, 011) would be parsed strictly as octal numbers, returning parse errors for 08 and 09 (invalid octal digits), and interpreting (010, 011) as octal numbers with decimal values (8, 9).

(*New in 12.2S, 12.4, 12.4T with CSCsc84077*) Octal parsing is removed from the NUMBER and NUMBER-related macros mentioned in this section, and the NUMBER_STRICT flag has been removed from the code entirely. These macros now have a strict interpretation of hex and decimal numbers as follows:

- A number string that begins with 0x or that contains the hexadecimal digits [A|a] through [F|f] is parsed as a hex number. The parse fails if the string contains an invalid hex digit.
- All other number strings are parsed as decimal numbers. The parse fails if the string contains a non-valid decimal digit.

In this case, the series of numbers (08, 09, 010, 011) would be parsed strictly as decimal numbers, with decimal values (8, 9, 10, 11).

To parse octal numbers with this implementation, you can use one of the following methods:

- Use the OCTAL or IOCTAL macro. This is the preferred method. In this case, the parse fails with input value 08 (invalid octal digit), and parses 010 or 10 as octal, with decimal value (8).

- Use the `GENERAL_NUMBER` macro (and `GENERAL_CHAR_NUMBER` or `GENERAL_CHAR8_NUMBER` for numeric character parsing) with the `NUMBER_OCT` | `OCT_ZERO_OK` flags to parse leading-0 numbers as octal. Add the `OCT_NO_ZERO_OK` flag to parse non-leading-0 numbers as octal. Depending on the `OCT_*` flag settings, this case parses 010 or 10 as octal, with decimal value (8).
- (*Not recommended*) If required only for backwards compatibility, use `GENERAL_NUMBER` with the new `OCTAL_NOT_STRICT` flag to replicate the earlier fall-through mixed parsing of octal and decimal numbers first described in this section. For this behavior, set the following flags in the `flags` parameter passed to the `GENERAL_NUMBER` macro:

```
NUMBER_OCT | OCT_[NO_]ZERO_OK | NUMBER_DEC | NUMBER_HEX |
HEX_[NO_]ZEROX_OK | [NUMBER_[NO_]WS_OK] | OCTAL_NOT_STRICT
```

See `NUMBER_*` flags and `*_NUMBER` flags in the *CISCO IOS API Reference Guide* for more information on the flags that specify how numbers are parsed.

27.3.3.2 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. `&enable_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,
    &enable_level
};
```

27.3.3.3 Example: Parse a Number from a Discontiguous Range

To parse a number within a discontiguous range, you can use a set of `NUMBER` macros to setup individual ranges. The following example shows how to parse a number within a set of a discontiguous range of numbers of 0, 1-2, 8, 62-63:

```
EOLNS(num_eol, num_cmd_handler);

NUMBER(num_range_4, num_eol, no_alt,
      OBJ(int,1), 62, 63, "Number Range 4");
NUMBER(num_range_3, num_eol num_range_4,
      OBJ(int,1), 8, 8, "Number Range 3");
NUMBER(num_range_2, num_eol, num_range_3,
      OBJ(int,1), 1, 2, "Number Range 2");
NUMBER(num_range_1, num_eol, num_range_2,
      OBJ(int,1), 0, 0, "Number Range 1");
```

The help string for the above example displays as follows:

```
<0-0> Number Range 1
<1-2> Number Range 2
<62-63> Number Range 3
<8-8> Number Range 4
```

27.3.4 Parse a Keyword-Number Combination

A keyword followed by an integer value is a common occurrence in 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)
```

27.3.4.1 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_no, snark_command, SNARK);
KEYWORD(snark_kw, snark_nvgen, ALTERNATE, "snark", "Snark command",
PRIV_ROOT);
```

27.3.5 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);
```

27.3.6 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.

27.3.6.1 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)
 */

EOLS      (exec_ping_eol, ping_command, 0);

STRING   (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);
```

27.3.7 Negating Commands — the “No” and “Default” Keywords

In configuration mode, most commands can be “negated” by prefixing them with the keyword **no**. Many commands can also be reset to the system default setting by prefixing them with the keyword **default**. **default** should map to either a particular configuration setting or its “negation”, varying per command.

The parsing of the **no** and **default** keywords is built into individual parse trees via the inclusion of the **NO_OR_DEFAULT** macro that is typically added by the tree’s creator for the future benefit of all other chain developers. It looks for these keywords and pops them from the command line before it even begins following the parser chains. The **NO_OR_DEFAULT** macro is typically placed near the root of the parse tree for early processing by the parsing engine.

When defining new commands, you do not create any keyword nodes for these keywords—your commands can automatically be entered in negated form; however, the parse tree should already provide support for the “**NO_OR_DEFAULT**” node at the first level alternate path.

Although you do not need to add nodes to *parse* these keywords, when defining commands you do need to add code that *recognizes* when the keywords have been used and reacts appropriately. Otherwise, your **xxx**, **no xxx**, and **default xxx** commands will end up doing precisely the same thing!

Note The **no** and **default** keyword prefixes are not allowed in EXEC mode, with the single exception of the debug parse chain (which has approved support for the **no** keyword only). This requirement helps catch developers who wrongly attempt to put configuration commands into EXEC mode, in violation of IOS CLI design requirements. Debug commands are special because they behave like configuration commands (that is, the operational state can be turned on/off), but they are temporary settings that are excluded from the configuration.

The creator of a new configuration subtree would normally add support for the **no** and **default** prefixes by including a “**NO_OR_DEFAULT**” node in the first level alternate path. In rare cases where it was not added by the creator, it may be appropriate to add it when needed, assuming the developer takes care to locate and remove any “**no**” KEYWORD entry that might be in place to provide partial (typically “**no**” only) prefix support.

Note Sometimes, developers may think that the **no** keyword in EXEC mode is required, when in fact **clear** is the appropriate root keyword. For example, in the case of the **clear ip arp address** command, the configuration is not removed as such; instead the arp table relearns the IP address when necessary.

27.3.7.1 “No” versus “Default”

Historically, IOS had only the **no** prefix. The **default** prefix is a relatively recent addition to IOS.

The original intended purpose of the **no** prefix was to erase a command from the configuration. That is, **no xxx** meant “remove the command **xxx** from the config.” Since IOS will assume a default value for any attribute which is not explicitly specified, **no xxx** also implicitly had the connotation “restore the value of **xxx** to its default.”

Under this interpretation, one would never expect to see **no xxx** generated in response to commands like **show running-config**, since another guideline within IOS is that commands are not generated in configurations when they are at their default value.

Unfortunately, the word **no** was perhaps not the best choice for this purpose. IOS configuration is not English, but users will tend to read it that way, and it is fairly natural for an English speaker to read **no xxx** not as “remove all mention of xxx” but rather as “opposite of xxx” or “the value of the parameter xxx is nil.” Some programmers have gone the same route when implementing IOS commands.

Consider for example the **ip address** command, as in:

```
interface Serial0
    ip address 192.2.3.4 255.255.255.0
```

This command is used to set a parameter whose value is a tuple of *<IP Address, Subnet Mask>*. In negative form, the command may be given as **no ip address 192.2.3.4 255.255.255.0**, which can also be reduced to **no ip address** (see section 27.3.7.4 “Parameter Elision in Negative Forms - the NOPREFIX Macro.”) But what exactly does this mean?

Under the interpretation of **no** as “erase,” it would mean that the entire command would be wiped from the configuration, and IOS would assume a default value for the IP address. That default *might* be “no IP address at all; IP is disabled,” but other choices such as “use IP unnumbered mode, adopt the address of the first LAN interface in the box” are equally valid.

In fact, the programmers that implemented this particular command did not treat **no** as meaning “erase,” but rather went with the alternate interpretation of “has a value of nil.” The command **no ip address** is interpreted to mean “interface has no IP address, and IP is disabled here.” This isn’t even regarded as a default case¹, and a command like **show running-config** will indeed generate output like:

```
interface Serial0
    no ip address
```

This particular oddity could have been avoided by implementing a slightly different syntax to indicate the lack of an IP address, for example **ip address none**, or that IP was disabled altogether, for example, **ip disabled**. But that’s not how it was done.

This ambiguity occurs particularly frequently with commands which represent Boolean parameters. Most Boolean commands do not include an explicit indication whether the parameter is on or off; the mere presence of the command in the config is taken to mean that the parameter is on, and the absence of the command implies the parameter is off. For example, whether or not the IP code will accept an all zero’s value in the IP subnet field in an IP datagram is controlled with the following Boolean command:

```
ip subnet-zero      ! Allow all zero subnet field
```

When the default for a Boolean option is that it is off, then it is immaterial whether **no xxx** removes the command from the configuration (and the system uses the default of off), or whether this explicitly sets the value of the parameter to off. The end result is the same. However, if the default for a Boolean command is that it is on, then it becomes confusing as to whether **no xxx** wipes the command from the config (and an ‘on’ value is assumed), or sets the parameter off. The majority of Boolean commands in IOS are implemented in the latter mode, and it is quite common to see the negative forms generated in the configuration.

This double-meaning for the **no** keyword leads to a problem. How do you reliably set a value back to its default, and erase it from the config, when **no xxx** can no longer be guaranteed to do this?

The **default** keyword was added to IOS to address this problem. Using a command of the form **default xxx** should *always* revert the parameter to its default value. As the command is now at its default value, it *should not* be generated in response to commands like **show running-config**.

1. Actually, for this particular command, this *is* the default, so explicit generation of the negative form in the configuration can be regarded as a minor bug. We ignore that detail for the sake of the discussion.

Note There is one exception to the rule that commands should not be generated when they are at their default values. If the default value is changed between one IOS release and the next, then the value should be explicitly generated even when at the default, up until the next major release. This allows a measure of backward compatibility—if the user should save their configuration using the new release, they can back up one level and not have unexpected side effects because the absence of the command leads to a different default being used.

As far as the end user goes: if the default for a command **xxx** is **no xxx**, then it is immaterial whether they type **no xxx** or **default xxx**, as the end result is the same. However, if the default is not **no xxx**, then obviously **no xxx** and **default xxx** will have two different results.

27.3.7.2 Handling the Negative Forms of Commands

As noted in section 27.3.7 “Negating Commands — the “No” and “Default” Keywords,” the parser automatically takes care of parsing the no and default keywords at the front of commands, but it is up to the programmer to add the code necessary to make your command do something intelligent when a negative form is used.

Whether either of these keywords was specified may be determined by looking at two flags within the parse context structure (the “csb”). These are the `csb->sense` and `csb->set_to_default` flags. The rules are:

- If *either* of the **no** or **default** keywords is parsed, `csb->sense` is set to FALSE. Otherwise it is set to TRUE. So `csb->sense` distinguishes the negative forms from the positive form.
- If the **default** keyword is parsed, `csb->set_to_default` is set to TRUE, else FALSE.

As long as the default for a command **xxx** is **no xxx**, it is unnecessary for you to examine the `csb->set_to_default` flag. Checking the `csb->sense` suffices.

Within the code, there are two possible ways to handle these flags. One is to add nodes directly into the parser chain which test the `csb->sense` or `csb->set_to_default` flags, leading to multiple paths through the parser. Though rare, constructs like the following are possible:

```
EOLNS(pos_node, do_something);
EOLNS(neg_node, do_something_else);
IFELSE(start_node, pos_node, neg_node, csb->sense);
```

Far more commonly, the parser chain will be relatively insensitive to these flags, and the flags are tested in the function that is ultimately called when the command is successfully parsed:

```
/*
 * aaa new-model
 */
EOLS(cfg_aaa_eol, aaa_command, AAA_NEW_MODEL);
KEYWORD(cfg_aaa_new, cfg_aaa_eol, no_alt,
        "new-model", "Enable NEW access control commands and functions.",
        PRIV_CONF);
KEYWORD(cfg_aaa, cfg_aaa_new, ALTERNATE,
        "aaa", "Authentication, Authorization and Accounting.",
        PRIV_CONF);

void aaa_command (parseinfo *csb)
{
    ...
    switch (csb->which) {
        ...
}
```

```

        case AAA_NEW_MODEL:
            aaa_new_model = csb->sense;
            break;
    }
}

```

27.3.7.3 Difference between NONE and no_alt

In non-componentized Cisco IOS branches, the `no_alt` value tells the parser that it is the last element in a chain of alternates. The `no_alt` value then checks for problems such as ambiguous commands. However, in componentized branches, a facility was developed to automatically detect the last element in a chain of alternates. So, in the componentized branches, `NONE` and `no_alt` are identical and you can use either one.

You can tell whether a branch is componentized by looking for ‘`parser@something`’ in the `/vob/ios/.publication_manifest` file. If ‘`parser@something`’ is available in the file, then the branch has a parser component in it.

27.3.7.4 Parameter Elision in Negative Forms - the NOPREFIX Macro

As a general rule in IOS, when a command syntax includes an explicit value at the end of the command, it is preferred that the user not be required to type the value when the negative form of a command is used, especially as the value rarely has any meaning in this context. For example, with the command `ip address 192.2.3.4 255.255.255.0`, it is desirable to be able to reduce the negative form of the command to `no ip address` (or `default ip address` as the case may be).

This is the one case where the `csb->sense` flag is commonly tested directly in the parser chains. By checking the `csb->sense` flag, just after parsing the keyword **address**, it is possible to skip over the parsing of the address and subnet mask when the command is given in negative form.

In fact, there is a special parser macro designed for just this purpose. The **NOPREFIX** macro tests the `csb->sense` flag and immediately jumps to its “accept” node if the `csb->sense` is FALSE. The **NOPREFIX** node also skips over any remaining text on the command line before jumping to the “accept” node (therefore it doesn’t matter what you type after that point on the command line, that text will be ignored).

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** or **default** version of this command, anything entered after the keyword **zip-query-interval** keyword is ignored.

```

/*
 * appletalk zip-query-interval <interval>
 * no appletalk zip-query-interval [<interval>]
 *
 * OBJ(int,1) = interval
 */
EOLS      (cr_at_zonequery_eol, appletalk_command, ATALK_ZONEQUERY);
NUMBER    (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);
KEYWORD   (cr_at_zonequery, cr_at_nozonequery, cr_at_arp,
           "zip-query-interval", "Interval between ZIP queries", PRIV_CONF);

```

Note NOPREFIX tests the `csb->sense` value, but not the `csb->set_to_default`. Therefore it will terminate a command at that point if *either* **no** or **default** is used.

27.3.8 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.

Note The parser NV generation code always walks the entire config mode parse tree, so you need to introduce the `NVGEN` parser node before other nodes, such as the `NUMBER` parser node, to avoid errors.

27.3.9 How to Block Unsupported Commands

It has become increasingly frequent for a customer to hear that a particular feature, feature-platform combination, or combination of features is NOT SUPPORTED. The Feature Navigator tool now Software Advisor on Cisco.com provides some cross-platform, cross-image feature support information, but it's mostly useful for “big” features and platforms and does not tackle the cross-feature supportability. It should be made quite apparent for customers to know if a specific feature is supported in their platform; Cisco IOS should provide the mechanisms for a customer to know if a feature is supposed to work. This section does not determine who defines what is and what is not supported. It just provides some coding “Best Practices” about how to do the right thing.

27.3.9.1 Definitions of “Unsupported”

The term “unsupported” is often made synonymous to “not tested.” This practice has been taken to extremes in cases where platform-independent control-plane features are labeled “unsupported” in some platforms because they haven't been tested. The universe of permutations to theoretically test is huge. For some customers, unsupported means that Cisco will not fix bugs. It's GOOD to leave things supported if they MIGHT work. The goal is to reach the point where “unsupported” means we know it WON'T work. If you can configure it, it's supported.

27.3.9.2 Best Practices

The good news is that there are coding best practices that can significantly alleviate the problem. Eight different best practices are presented:

- Images
- Subsystems
- Parser
- Registry
- Command Line
- PRIV_HIDDEN
- PRIV_UNSUPPORTED
- Documentation

27.3.9.3 Images

Ensure that all images that are posted to Cisco.com are “supported” by the respective platform. This pertains to IOS trains as well as feature sets. For example:

CSCdz09446 RPM crashes upon loading IOS full image 12.2(12)

It turns out that `f1o_t` images (12.2M) are not “supported” by the RPM platform. However, they were built and posted on Cisco.com.

This defect is closed with the following C-comments:

“RPM uses the images from the `f1o_t` branch (T train images), so it is recommended to use the official release 12.2-11.T. Looking at the care-details, it is found that the customer was asked to upgrade the 7200 router (connected BPX) to the 12.2.12 image and because RPM was present on another site, the customer tried to upgrade RPM with the same image. As per the 7200 team, they didn’t see this problem with the 12.2.12 image (correct image for 7200). For RPM, please use T train images. I’ve tried upgrading the RPM with 12.2.11T and 12.2.12T images, with which I didn’t see this problem. Hence I’m closing this DDTs. At the end, 12.2M and 12.3M images for the RPM platforms (`rpm-* -mz` and `rpmpxf-* -mz`) were deferred; CSCeb31735 is the defect needed for the deferrals.”

27.3.9.4 Subsystems

Make sure that you include subsystems only in images/platforms that support the feature. This best practice also relates to image sizes. For example:

- CSCee34514 `bba_group` subsystem should be removed from `ipbase` and `entbase` images

The `bba_group` subsystem contains the CLI for configuring PPPoE server, but the `pppoe_server` subsystem contains the code that actually implements the feature. This packaging combination allows customers to configure the PPPoE feature in images where the feature is not present and this is very confusing.

- CSCdz25609 Remove VPDN support in 12.2S

The entire VPDN feature set is not being marketed in 12.2S. If we’re not marketing it, we shouldn’t include it. If it’s present in the image, somebody may actually try to use it.

When removing a subsystem from an image, make sure it doesn’t break functionalities that we do support via registry calls to the removed subsystem. As an example, CSCee52889 is a regression to CSCdz25609 because `vpn_select_tas()` registry decodes both VPDN and IPSec services; removing the VPDN subsystem causes the registry to return `default_reg_return_0`, although that is also used by IPSec.

- CSCee23329 CAT6500/7600 Removal of ISDN subsystem from the image
Removal of ISDN code in 7600, since it is not supported.
- SOHO platforms also include cxxx_remove_cli subsystems (xxx being the platform number) to hide unsupported commands.

For example:

- c870_remove_cli.c—Subsystem that is used to remove some unwanted CLI commands from c870 platform images.
- CSCef03197 Add keyword blocking support to block PLUS features
- CSCdy03593 Remove unsupported options for Squeeze functionality

27.3.9.5 Parser

Use IFELSE/ASSERT or TEST_EXPR/TESTVAR/TEST_* parser macros to hide commands when “unsupported.” For example:

The xconnect code uses IFELSE to check for platform capability/supportability for each of the options in the command including the topmost xconnect CLI.

Here is a snippet from xconnect/cfg_int_xconnect.h:

```

KEYWORD_ID (cfg_xconnect_sequencing_both, cfg_xconnect_eol, no_alt,
    OBJ(int, 4), PW_SEQUENCE_BOTH, "both",
    "Transmit and receive sequence numbers", PRIV_CONF |
    PRIV_SUBIF);

IFELSE (cfg_xconnect_sequencing_both_test,
    cfg_xconnect_sequencing_both,
    cfg_xconnect_eol, xconnect_supports_sequencing(csb,
    PW_SEQUENCE_BOTH));

KEYWORD_ID (cfg_xconnect_sequencing_recv, cfg_xconnect_eol,
    cfg_xconnect_sequencing_both_test, OBJ(int, 4),
    PW_SEQUENCE_RECV,
    "receive", "Receive sequence numbers", PRIV_CONF |
    PRIV_SUBIF);

IFELSE (cfg_xconnect_sequencing_recv_test,
    cfg_xconnect_sequencing_recv,
    no_alt, xconnect_supports_sequencing(csb, PW_SEQUENCE_RECV));

KEYWORD_ID (cfg_xconnect_sequencing_xmit, cfg_xconnect_eol,
    cfg_xconnect_sequencing_recv_test, OBJ(int, 4),
    PW_SEQUENCE_XMIT,
    "transmit", "Transmit sequence numbers", PRIV_CONF |
    PRIV_SUBIF);

IFELSE (cfg_xconnect_sequencing_xmit_test,
    cfg_xconnect_sequencing_xmit,
    cfg_xconnect_sequencing_recv_test,
    xconnect_supports_sequencing(csb, PW_SEQUENCE_XMIT));

KEYWORD (cfg_xconnect_sequencing, cfg_xconnect_sequencing_xmit_test,
    cfg_xconnect_eol, "sequencing",
    "Configure sequencing options for xconnect",
    PRIV_CONF | PRIV_SUBIF);

```

```

IFELSE (cfg_xconnect_sequencing_test, cfg_xconnect_sequencing,
    cfg_xconnect_eol,
    (xconnect_supports_sequencing(csb, PW_SEQUENCE_XMIT) ||
     xconnect_supports_sequencing(csb, PW_SEQUENCE_RECV)));

[snip]

KEYWORD_ID (cfg_xconnect_encap_mpls, cfg_xconnect_pw_class_opt,
    no_alt, OBJ(int, 2), PW_ENCAP_MPLS,
    "mpls", "Use MPLS encapsulation", PRIV_CONF | PRIV_SUBIF);

IFELSE (cfg_xconnect_encap_mpls_test, cfg_xconnect_encap_mpls,
    no_alt, xconnect_atom_supported_on_interface(csb));

[snip]

KEYWORD_ID (cfg_xconnect_encap_l2tp,
    cfg_xconnect_encap_l2tp_manual_set,
    cfg_xconnect_encap_mpls_test, OBJ(int, 2),
    PW_ENCAP_L2TPV3,
    "l2tpv3", "Use L2TPv3 encapsulation", PRIV_CONF | PRIV_SUBIF);

IFELSE (cfg_xconnect_encap_l2tp_test, cfg_xconnect_encap_l2tp,
    cfg_xconnect_encap_mpls_test,
    xconnect_l2tp_supported_on_interface(csb));

[snip]

KEYWORD (cfg_xconnect_vfi, cfg_xconnect_vfi_name,
    cfg_xconnect_peer_id,
    "vfi", "connect to a virtual forwarding instance",
    PRIV_CONF | PRIV_SUBIF);

IFELSE (cfg_xconnect_vfi_test, cfg_xconnect_vfi, cfg_xconnect_peer_id,
    xconnect_supports_vfi(csb));

[snip]

KEYWORD_MM (cfg_int_xconnect, cfg_xconnect_noprefix, ALTERNATE,
    "xconnect", "Xconnect commands", PRIV_CONF | PRIV_SUBIF, 2);

IFELSE (cfg_int_xconnect_test, cfg_int_xconnect, ALTERNATE,
    xconnect_supported_on_interface(csb));

```

Also note that the **IFELSE** check function ends up invoking a registry that platforms can add to. For example, for checking for sequencing, `reg_invoke_l2tun_switching_capabilities()` and `reg_invoke_atom_hwidb_capability()` are performed. For checking for AToM support, `reg_invoke_atom_imposition_capable()` is called, and for checking for L2TPv3 support, `reg_invoke_l2tun_interface_capabilities()` is invoked.

Another example would be the addition of the “l2transport” keyword to an existing command using the `ASSERT` macro.

Here is a snippet from `wan/cfg_conn_pw_seg2.h`:

```

KEYWORD (fr_pw, fr_pw_set_connection_type, no_alt,
    "l2transport", "Layer 2 packet over pseudowire config commands",

```

```

    PRIV_CONF | PRIV_SUBIF);

ASSERT (fr_pw_assert, fr_pw, no_alt,
        (GETOBJ(idb,1) &&
         is_frame_relay(GETOBJ(idb,1)->hwptr) &&
         (subsys_find_by_name("generic_atom") ||
          subsys_find_by_name("xconnect"))));

```

27.3.9.6 Registry

A feature can add a registry, to which platforms can add themselves in order to return their platform and interface capabilities. That is, platforms/interfaces that support the feature can add themselves to the registry and return TRUE, while platforms where the feature is not supported would not reg_add or would return FALSE. This best practice could be combined with the parser to hide commands where unsupported. (See section 27.3.9.5, “Parser.”)

An example of this is AToM code that creates the atom_hwidb_capability. Platforms can reg_add to supply their feature-support for specific hwidbs.

Here is some example code from AToM in 12.0S:

```

atom/atom_c7100.c:229:
reg_add_atom_hwidb_capability(atom_c7100_hwidb_capability,
atom/atom_les.c:54:
reg_add_atom_hwidb_capability(atom_les_hwidb_capability,
atom/atom_rsp.c:241:
reg_add_atom_hwidb_capability(atom_rsp_hwidb_capability,
src-bfrp/atom_bfr.c:47:
reg_add_atom_hwidb_capability(bfrp_atom_hwidb_capability,
toaster/camr_rp/camr_rp_eompls.c:895:
reg_add_atom_hwidb_capability(camr_atom_hwidb_capability,

```

The same example is followed in the 12.2S platforms as shown:

```

atom/atom_c7100.c:226:
reg_add_atom_hwidb_capability(atom_c7100_hwidb_capability,
atom/atom_c7300.c:472:
reg_add_atom_hwidb_capability(atom_ws_hwidb_capability,
atom/atom_les.c:60:
reg_add_atom_hwidb_capability(atom_les_hwidb_capability,
atom/atom_rsp.c:245:
reg_add_atom_hwidb_capability(atom_rsp_hwidb_capability,
const/common-rp/cwan_mpls.c:1718:
reg_add_atom_hwidb_capability(cwan_atom_hwidb_capability,

```

In fact, the support of subfeatures for specific interfaces can be seen with the exec command **show mpls l2transport hw-capability**. Please refer to `atom_show_mpls_l2_cap()` in `atom/atom_parser.c`.

The function `atom_c7100_hwidb_capability()` unfortunately leaves a back door for unsupported capabilities to be configured. We can see from the following code snippet that `atom_c7100_hwidb_capability()` would indeed return capabilities if the **service internal** command has been configured. Although **service internal** is a `PRIV_HIDDEN` command, many customers do have **service internal** enabled and it is documented externally as well as on Cisco.com.

```

/* Disable support for Non-VXR */
switch (c7100_chassis_version & C7100_MP_VERSION_MASK) {
    case C7100_MP_VX_VERSION:

```

```

        case C7100_MP_21554_VERSION:
            break;

        default:
            if (!internal_cmd_enable) {
                return (cap);
            }
            break;
    }

```

In link-bundle interfaces in GSRs (`if/if_lb_feature_support.c`), registries are also used to test feature-support.

Here is a code snippet from `if/if_lb_feature_install.c`:

```

/*
 * Tries to get the feature cb for a feature.
 * Only if the platform supports the feature, this call will return
 * valid feature cb
 */
lb_feature_cb_t *
lb_get_feature_cb (lb_media_type media_type, lb_feature_t feat)
{
    lb_feature_platform_info *fpi;

    /*
     * we return the feature cb only if the platform supports it
     */
    fpi = lb_get_feature_platform_info(media_type, feat);
    if (!fpi || !fpi->feature_supported) {
        return NULL;
    }

    return reg_invoke_lb_get_feature_cb(feat);
}

```

The following two defects show how features add themselves to the `lb_get_feature_cb` registry. Specifically:

- CSCef14040 for AToM feature
- CSCef14040 LB-Phase3: Feature Blocking support for AToM

```

void
lb_atom_init (void)
{
    reg_add_lb_get_feature_cb(LB_FEATURE_ATOM,
                             lb_get_atom_fcb,
                             "lb_get_atom_fcb");

    ...
}

```

Here is a code snippet that illustrates CSCin78990 feature control block support for MQC:

```

void
lb_qos_init (void)
{
    reg_add_lb_get_feature_cb(LB_FEATURE_QOS,
                             lb_get_qos_fcb,
                             "lb_get_qos_fcb");
}

```

In the `cxxx_remove_cli` subsystem initialization for SOHO platforms, a callback has been added for the `is_keyword_supported` parser registry to block certain unsupported keywords.

Here is a code snippet from `src-mpc-c800/c800_remove_cli.c`:

```
/*
 * c800_remove_cli_init()
 */
void c800_remove_cli_init (subsstype *subsys)
{
    reg_add_is_keyword_supported(0,
                                is_keyword_supported_for_c800,
                                "is_keyword_supported_for_c800");
    reg_add_default_is_c800_image(is_c800_image,"is_c800_image") ;
}
```

And then the `is_keyword_supported` registry is invoked from `match_partial_keyword()` in `parser/parser_actions.c`. This is a RETVAL registry with a Boolean return.

Here is a code snippet from `parser/parser_registry.reg`:

```
DEFINE is_keyword_supported
/*
 * Check if entered CLI keyword is supported
 * on given platform
 * Return: TRUE/supported FALSE/not supported
 */
RETVAL
boolean
const char *keyword
0
uchar method
END
```

Here is a code snippet from `src-mpc-c800/c800_remove_cli.c`:

```
boolean is_keyword_supported_for_c800(const char *keyword)
{
boolean retval = TRUE;
int cnt;
int number_of_keywords;
number_of_keywords = (sizeof c800keywords) / (sizeof
*c800keywords);
for (cnt = 0; cnt < number_of_keywords; cnt++)
{
    if (strcmp(keyword, c800keywords[cnt]) == 0)
    {
        /*
         * entered keyword matches keyword in
         * table so should block it for C80X
         */
        retval = FALSE;
        cnt = number_of_keywords;
    }
}
return (retval);
}
```

Note that by default, this registry returns true.

Here is a code snippet from `parser/chain.c`:

```
/*
 * This is the default function for the parser registry:
 * is_keyword_supported.
 *
 */
boolean is_keyword_supported(const char *keyword)
{
    return (TRUE);
}

void parser_add_main_modes (void)
{
[snip]
/*
 * Add default function for parser registry: is_keyword_supported.
 */
reg_add_default_is_keyword_supported(is_keyword_supported,
"is_keyword_supported");
}
```

Note that this function will block the use of the specified keyword(s) in any position in commands. For example, when blocking “eigrp” and “aaa,” no commands including the keywords “eigrp” or “aaa” can be issued. This is not a very fine-grained method for disabling commands, therefore it is preferable to use a more exact method to disable commands where available.

To block any unsupported commands for your specific platform, use a registry function so that the unsupported commands will not be visible to the user in the CLI or in the help. A number of parser-macro-interpretation routines have imbedded platform-specific tests to exclude particular options or commands or to query platform configuration. To do this a number of registry services have been established for use by your platform. For example, three such registries are:

- `reg_invoke_isc5rsp()`

This registry returns TRUE when this IOS is supporting a Cat5k “Yosemite” RSP L3 switching module.

This is a STUB registry with a Boolean return. If no routine is registered, the STUB support code automatically returns FALSE.

Using a function such as this is not the best solution. For example, if three different platforms did not support the **foo** command: the Cat5k RSP, the Cat6k (which supports a hypothetical `reg_invoke_is_cat6000()` function) and the Cat4k (which supports a similar hypothetical function), the **foo** command would then need to check for, say,

```
reg_invoke_isc5rsp() && reg_invoke_is_cat6000() && reg_invoke_is_cat4000()
```

This means that the file defining the **foo** chain will need to be modified every time another platform doesn’t support the **foo** command, which is not very flexible. It would be better for a `reg_invoke_is_foo_unsupported()` LOOP registry to be added, to which platforms that do not support **foo** can add a function that returns TRUE; the default return of a registry is FALSE so if no function is added to the registry by the platform, **foo** will be available.

This method is extensible to cases where **foo** is an interface-level command that is only available on some interfaces; the registry could take the hwidb as a parameter and platforms could register functions with the logic they require to determine whether **foo** is acceptable on a particular interface.

- `reg_invoke_is_qos_int_cli_disabled()`

This registry is an example of a good way of blocking commands as opposed to adding calls to `reg_invoke_is_xyz_platform()` everywhere. This registry returns TRUE if the legacy (non-MQC, interface-level) QoS command indicated by `the_cli` is not supported on the interface indicated by `hwidb`. Commands such as “fair-queue,” “traffic-shape,” “custom-queue-list,” etc. can be blocked using this registry. See CSCdw39321 for details.

- `config_is_cli_disallowed()`

A new CASE_LOOP registry service, `config_is_cli_disallowed()`, has been added to 12.0S, 12.2S, and 12.3T. The purpose of this service is to perform a compatibility check of a configuration command being entered against the current configuration. The command can be accepted or rejected based on the current configuration. Although platform registries are available to block CLIs not supported on certain platforms, this new registry can be used not only for platform commands but also for detecting mutually exclusive commands to avoid ambiguous configuration.

Another difference is that this registry is called in the parser action routine of the command that desires to perform the check. If some incompatibility is detected, the command will be rejected and an error message printed.

The following is the definition of the new registry:

```
/*
 * Used by config_is_cli_disallowed registry for specifying a command
 * category.
 */
typedef enum cfg_elem_type_ {
    CFG_ELEM_IP_ADDR,
    CFG_ELEM_XCONNECT,
    CFG_ELEM_MAX,
} cfg_elem_type;

DEFINE config_is_cli_disallowed
/*
 * Summary: Check if a CLI command should be blocked.
 * Inputs: Command category, swidb, and command category dependent data.
 * Returns: TRUE if the command is not allowed.
 */
CASE_LOOP
boolean
idbtype *swidb, void *data
CFG_ELEM_MAX
cfg_elem_type type
END
```

A good rule of thumb to keep in mind when blocking unsupported commands is to ask “does the platform have X *capability*,” not “is the platform of *type* Y.”

The following statements are true for registry functions:

- Each registry function needs to be as efficient as possible because it is called for each command in the configuration file. Remember also that registry functions are used for a variety of purposes, many of them unrelated to parser functions. In any case, code should be as efficient as possible.
- A general truism for all loop type registries is the following—filter unrelated commands (non application specific stuff) at the start of the registry function only. This means that first task within the registry routine should be to return if this is not a command you are handling.

- A general truism unless the “past invocations or state” are not part of the general system state is the following—the registry function should make decisions based on the arguments *csb* and *idb* that are passed in and the present system state. Relying on the past invocations or state is inefficient and prone to problems.
- To denote success but no command sync using the PRC phase III API, use the following code:

```
/* Get a pointer to the PRC data block from the csb */
prc_t prc = prc_get_block_ptr(csb);

/* Set the success 'bit' in the PRC data block */
prc_success(prc, ...);

/* Disable the syncing of this command to the standby */
prc_ha_set_sync_enable(prc, PRC_HA_DONT_SYNC);
```

If a command that you believe is valid is not working on your platform, first check whether your platform has registered for any of the platform-exclusion services, such as those listed above.

If you want to exclude a command on your platform, register your routine with the appropriate platform-exclusion service, such as those listed above. Call upon the parser-questions@cisco.com alias for help.

If you create a new service, please notify ios-doc@cisco.com so that the above list can be augmented.

27.3.9.7 Command Line

Sometimes it does not make sense to hide commands from the CLI. One example is the MQC code in which class-maps and policy-maps are created in advanced and later on applied as a service-policy in an interface. Therefore, there's no way to know beforehand to which [sub]interface the service-policy will be applied. However, MQC code provides “not supported” checks at the time of installing the service-policy. See the QoS registry `qoscli_policy_install_filter_check`. Here is an example from `qos/qos_chain.c`:

```
reg_add_qoscli_policy_install_filter_check(MATCH_ATM,
    match_atm_install_check, "match_atm_install_check");
```

Here is an example from `qos/match_atm_chain.c`:

```
/*
 * match atm-clp is only allowed on ATM interfaces
 */
void match_atm_install_check (hwidbtype *hwidb, if_type mqc_iftype,
    void *if_info, filter_t *filter, uint dir, boolean
    *ok_to_install)
{
    match_atm_params_t *atm_params =
        (match_atm_params_t *)filter->match_params;

    if (dir != IN_POLICY && atm_params->match_atm_type ==
        MATCH_ATM_CLP) {
        *ok_to_install = FALSE;
        printf("\n'match atm clp' allowed only in an input policy");
        return;
    }
    if (!is_atm(hwidb)) {
        *ok_to_install = FALSE;
        printf("\n'match atm' supported only on ATM interface");
    }
}
```

```

        return;
    }

    if
    (!reg_invoke_qoscli_is_match_atm_supported(atm_params->match_atm_type,
                                                hwidb)) {
        *ok_to_install = FALSE;
        printf("\n'match atm %s' is not supported on this ATM
               interface",
               match_atm_print(atm_params));
        return;
    }
}

```

We can see that when this command is used, an error message is printed if the feature is not supported and `*ok_to_install` is set to FALSE.

Another example is illustrated when transparent bridging is configured. There are limitations in the driver code in the turbo switch with interfaces that are also configured for MPLS. Specifically, the “bridge-group” is not supported in MPLS interfaces and in ATM and Frame Relay point-to-point MPLS subinterfaces. Because it is “not supported” (that is, we KNOW it does not work), the configuration is prevented and a message is printed in the CLI.

The following example taken from `tbridge/tbridge_config.c` shows a `reg_invoke` for the `platform_hw_bridging_capable_if` registry, along with the checks for interfaces configured for MPLS:

```

void bridgegroup_command (parseinfo *csb)
{
[snip]
    static const char bridge_not_supported[] = "\nInterface has MPLS"
    " configuration\n%s cannot be configured into a Bridge
    Group";

    idb = csb->interface;
    hwidb = idb->hwptr;
[snip]
    if (reg_invoke_platform_hw_bridging_only() &&
        !reg_invoke_platform_hw_bridging_capable_if(hwidb)) {
        printf("\nBridging is not supported on this interface.");
        return;
    }
[snip]
    if (is_atm_swidb(idb) || is_fr_p2p_subint(idb)) {
        if (reg_invoke_mpls_configured_on_swidb(idb)) {
            printf(bridge_not_supported, idb->namestring);
            return;
        }
    } else {
        if (reg_invoke_mpls_configured_on_hwidb(idb->hwptr)) {
            printf(bridge_not_supported, idb->namestring);
            return;
        }
    }
}

```

The reverse is also checked with the following registry call:

Here is a code snippet from `tagsw_chain.c`:

```
if (reg_invoke_bridging_configured_on_idb(idb)) {
```

```

    if (cli_caller) {
        printf("\n%% MPLS not supported on interface %s"
              "\nTransparent bridging already configured",
              ldp_idb_get_namestring(idb, FALSE));
    }
    return (FALSE);
}

```

27.3.9.8 PRIV_HIDDEN

Do not use `PRIV_HIDDEN` to indicate that a command is unsupported. The only legitimate use for a *hidden* command is when it is being obsoleted. Using `PRIV_HIDDEN` to deprecate a command in favor of a new command allows migration to the new release without breaking any scripts.

Refer to section 27.11, “Guidelines for Internal and Hidden Commands and More,” for usage guidelines about `PRIV_HIDDEN`.

27.3.9.9 PRIV_UNSUPPORTED

Use (and don’t overuse) `PRIV_UNSUPPORTED`. The `PRIV_UNSUPPORTED` flag makes a command hidden, and also prints an error message as follows:

```

/*
 * Command isn't supported yet
 */
if (csb->unsupported && !csb->nvgem &&
    !system_loading && !bootstrap_enable &&
    (csb->flags & CONFIG_TERM)) {
    printf("\n%% This command is an unreleased "
          "and unsupported feature");
}

```

For example, when configuring a GRE Tunnel, two unsupported features are used to configure a bridge-group, and (as with all interfaces) are used with the command **shutdown force**:

```

Router(config)#interface Tunnel 0
Router(config-if)#bridge-group 1
% This command is an unreleased and unsupported feature
Router(config-if)#shutdown force
% This command is an unreleased and unsupported feature
Router(config-if)#

```

We can see the use of `PRIV_UNSUPPORTED` in those two parser macros.

Here is a code snippet from `srt/cfg_int_bridge-group.h`:

```

KEYWORD_MM(bgroup, bgroup_grp, NONE,
           "bridge-group", "Transparent bridging interface
                           parameters",
           PRIV_CONF|PRIV_SUBIF, 7);

PRIV_TEST(bgroup_unsupp, bgroup, NONE, NONE, PRIV_UNSUPPORTED);

TEST_IDBSTATUS(bgroup_unsupp_test, bgroup_unsupp, bgroup, NONE,
               IDB_TUNNEL);

```

Here is a code snippet from `parser/cfg_int_shutdown.h`:

```
EOLNS (ci_shutdown_func, shutdown_command);
```

```

KEYWORD_ID (shutdown_forced, ci_shutdown_func, ci_shutdown_func,
    OBJ(int, 2), TRUE,
    "forced", common_str_empty_str,
    PRIV_HIDDEN|PRIV_UNSUPPORTED);
KEYWORD_ID (shutdown_soft, ci_shutdown_func, shutdown_forced,
    OBJ(int, 1), TRUE,
    "soft", common_str_empty_str, PRIV_HIDDEN);
NVGENNNS (shutdown_nvgen, shutdown_soft, shutdown_command);
KEYWORD (shutdown, shutdown_nvgen, ALTERNATE,
    "shutdown", "Shutdown the selected interface",
    PRIV_CONF|PRIV_SUBIF);

```

This DDTs is another example: CSCed39001 Commit for banner for LC-ATM non-support.

This DDTs tracks the commit of diffs to display a banner when LC-ATM is configured on a box running RLS4 and later versions (until support for LC-ATM is provided). LC-ATM functionality should not be disabled as it is necessary for existing regression test beds.

Here are the functional requirements:

- When LC-ATM subinterface is configured, display a banner to the user:
%Warning! LC-ATM is not supported in this release.
- When the system is booted, display the same banner (intended to catch the case where LC-ATM is configured on earlier software, then RLS4 is loaded/booted afterwards).

27.3.9.10 Documentation

In the unlikely event that all of the above fails, document the “unsupported” combination on Cisco.com. There may be liability considerations in not doing so.

27.3.10 Generating Error Messages in Configuration Mode

By convention, all configuration command responses start with a \n. Therefore, if the output is dynamically generated by several different printf statements, none of the printf statements in the sequence need to concern themselves about whether they are the last statement of the message. Also, the \n at the beginning of the line ensures that each new line starts on the left margin.

Consider the following code:

```

printf("\n%% MPLS not supported on interface %s"
    "\nTransparent bridging already configured",
    ldp_idb_get_namestring(idb, FALSE));

```

Furthermore, you have no assurance that any message previous to your own has appended a \n.

For example, consider the configuration command response:

```
"% Interface foo does not support features: wobble wurtz wibble"
```

In this case, foo, wobble, wurtz and wibble can all be generated at different places in the code (for example, as part of a case statement) and still appear on a single line as long as none of the printf commands in the sequence add a superfluous \n.

By convention, responses to configuration commands always start out with a percent sign followed by a space (%). The percent sign (and space) leading a message indicates that you are looking at an error message generated by the system. This convention was patterned after the Digital Equipment

Corporation's TOPS-20 operating system. By adopting this convention, a transcript of a configuration session clearly distinguishes what was typed by the user and responses that were system-generated.

27.3.11 Configuration Editor in IOS?

You might have asked yourself why there is no configuration editor in IOS. Perhaps an embedded editor that could be used to make more complicated configuration changes when in configuration mode? Well, the running configuration isn't stored in memory like a file. It's a complex data store where every command/feature stores its configuration in its own way. If we allowed you to edit a configuration from A to B, we wouldn't be able to figure out which commands to issue to get from A to B. So, while maybe we could offer a **vi**-like editor, it would be useful for startup configuration, but not for running configuration. And it's a router, not a workstation, so we don't bother.

27.4 Adding Parser Return Codes

Removed subsections "27.3.4 Data Structures", "27.3.5 Algorithmic Description", "27.3.6 Interface Design", "27.3.9 Sample PRC Implementation", 27.3.10 Development Unit Testing", and "27.3.11 Migrating Obsolete PRC to Current PRC" when subsection "27.5.2 PRC Phase III and Action Function Return Codes" was added. (August 2005)

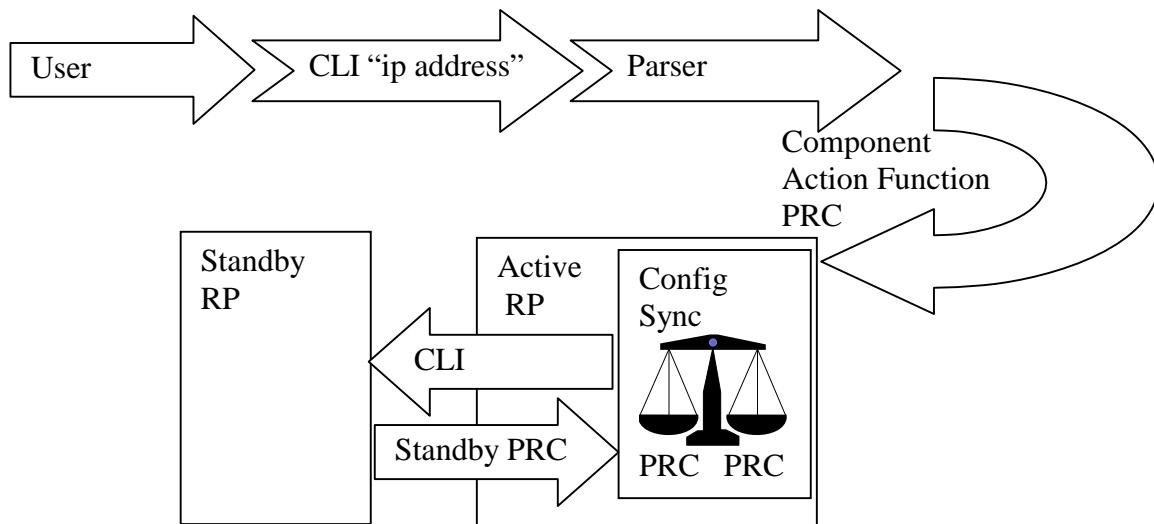
Parser Return Codes (PRC) are used primarily to signal the success or failure of a particular CLI command. Customers' scripts and test engineers' scripts can be modified to accept the Parser Return Code to detect the success or failure of the CLI command. By making use of the return code, customers and test engineers can greatly reduce the amount of time spent on developing customized applications to detect changes. It also increases the reliability of the applications when there is a return code provided for each command.

See section 27.5.2, "PRC Phase III and Action Function Return Codes" for the Hot ICE PRC information.

27.4.1 PRC Overview

Although PRC is useful on standalone systems, one of the first applications to make use of the PRC was the Config Sync code, which is part of the High Availability (HA) initiative. High Availability allows switchover from an Active Route Processor (RP) to a Standby RP in the event of failure. Whenever configuration commands are entered into the Active RP, they are also sent to the Standby RP, so that after a switchover the configuration remains the same. In order to sync the commands from the Active RP to the Standby RP, return codes are added to every IOS command. The return code specifies whether a particular IOS command was accepted or rejected. If the system behavior has been intentionally changed, then the return code indicates success.

Figure 27-4 shows the system flow when a command is entered. The command is entered from the user and goes through the parser to the component action function. The component action function returns a PRC to the Config Sync code on the active RP. The Config Sync code sends the CLI to the Standby, the standby runs the command and returns a PRC to the Active RP Config Sync code. Then, the Standby PRC is checked against the original PRC on the Active. If they both match, the Config Sync code knows that both RPs have the same configuration. If the PRCs differ, then the Config Sync knows that there is a problem syncing the command.

Figure 27-4 Example of PRC in HA

Another use for the PRC is to assist in doing configuration rollback. Rollback allows the customer to go back to a previously known working configuration.

Every new command in IOS needs to be created with PRC. In addition, every existing command within IOS needs to be modified to set a return code. The different IOS commands are responsible for setting the parser return code; however, there can be a period when commands have not yet properly set the parser return code. There is a transition period until all commands can be modified to set the return code. For this reason, there can be cases where the status is a legacy return code.

27.4.2 Terms

Action Function

A function that is part of an IOS component that is called when a line is successfully parsed.

Active RP

The RP that controls the system, runs the routing protocols, and presents the system management interface.

Blind Sync

A Blind Sync is a redundant system command synchronization method that sends the same command that was entered on the Active RP to the Standby RP regardless of whether the command execution is successful or not.

PRC

Parser Return Code - returned by an action function when done processing a command

RP

Route Processor.

Standby RP

The RP that waits for the active or primary RP to fail.

State

If an action function has multiple exit points that are considered successful, each of these are considered to be a separate state in PRC II. In PRC III, all such success states are considered as one.

Switchover

An event in which system control and routing protocol execution is transferred from a failed processor to a standby processor.

27.4.3 Functional Structure

The functional structure of PRC is described by explaining how the PRC is set, describing the PRC effects on users, describing action function success versus failure, and describing successful action function states.

27.4.3.1 How the PRC is Set

The major functionality of the PRC is implemented as part of the parser. The parser handles all incoming user CLI commands and invokes the appropriate action function. If the command fails at the parser level, for example because of a syntax error, the PRC will be set by the parser subsystem. If the command makes it through the parser and enters the action function, the action function should set the PRC. If the PRC is not set at the action function level, the PRC remains in an invalid state and it is identified as legacy code that did not implement PRC.

27.4.3.2 Users of PRC

From a user's perspective, the PRC is visible by invoking diagnostic commands (see End User Interface). For example, the HA Config Sync code makes use of the PRC to decide what to sync. HA code will be interested in checking the PRC in the success case, to verify that the action functions completed identically, with the same state. If the PRC indicates legacy return code, then the HA Config Sync code will use the *Blind Sync* method to sync the command to the Standby RP. In addition to this HA use, future customers' scripts and test engineers' scripts will depend on the PRC being returned through the CLI.

27.4.3.3 Partial Success is Complete Failure

For an action function to decide that a command was successful, the command must be a complete success without failure. Therefore, if a command has multiple compound actions, each action must not fail in order for the command to be considered successful. Commands must consider their results entirely successful or declare that the result is a failure, meaning the result of a command is binary: pass or fail. For example, HA will not synchronize a partial failure to the Standby RP.

27.4.3.4 Successful Action Function

If a command has more than one acceptable state upon completion (for instance, because it might perform different actions due to different conditions, but any of those different actions will be considered successful if they complete), it must return success code along with the change type (whether there was a config change) and change mode (the timing of the config change) as explained in section 27.5.2.

27.4.4 End User Interface

The end user interface is the **debug rc** command. This is an internal command, and it only becomes available after the user has entered the **service internal** config command:

debug parse rc

Or:

no debug parse rc

Following the execution of the **debug rc** command, the command exec shell will format and log the return code set by any CLI command.

Note A subsequent **debug rc** command supersedes a previous **debug rc** command.

Having instrumented a given command using the PRC Phase III API, the results can be tested by exposing the PRC data at the console. The **terminal prc expose** command will cause the parser to display the PRC data for each CLI before the next prompt. For example:

```
gt15-7200-2#terminal prc expose
gt15-7200-2(config)#logging console
!PRC[success, change-type=PRC_CONFIG_CHANGE, change-mode=PRC_IMMEDIATE,
ha-sync-enable=true][command=logging console ]
```

The PRC display can be disabled with the **terminal prc hide** exec command.

The config logger can also be configured to do PRC logging and to display the logger output, as shown here:

```
Router(config)#archive
Router(config-archive)#log config
Router(config-archive-log-cfg)#record rc
Router#show archive log config all
```

27.4.5 Restrictions and Configuration Issues with PRC

There are no restrictions or configuration issues due to PRC implementation for software and firmware. Any hardware capable of running IOS will be supported. Also, no external restrictions or configuration issues result from PRC implementation

27.5 Hot ICE

Hot ICE (Hot IOS Configuration Enhancements) is the first stage to provide a more robust and functional means of automated configuration and provisioning for IOS. Hot ICE includes support for Access Session Locking, BEEP/TLS, and HTTP/S. Hot ICE is supported by the following infrastructure features (*New in 12.5pi1, 12.2S, and 12.4T*):

- 27.5.1 Macro Modifications
- 27.5.2 PRC Phase III and Action Function Return Codes
- 27.5.3 Configuration Defaults Exposure
- 27.5.4 Rollback Component Conformance
- 27.5.5 Support for Syntax Checking
- 27.5.6 Configuration Change Notification
- 27.5.7 Before After API

The following subsections describe these infrastructure features, and they are followed by two subsections on using these features in IOS component development:

- 27.5.8 How to Instrument IOS Components for Hot ICE

- 27.5.9 Hot ICE Compliance Tooling

To develop Hot ICE components, go to the Hot ICE developer portal at:

<http://hotice.cisco.com>

27.5.1 Macro Modifications

One of the initial tasks for Hot ICE compliance is to change macro usage for several of the common macros. Table 27-5 lists the macros that must be converted and the macros that they should be converted to.

Table 27-5 New Macros for Hot ICE Compliance

	Deprecated Macro	New Macro
1	EOLNS	EOLNS_COMMAND
2	EOLS	EOLS_COMMAND
3	GENERAL_PARAMS	GENERAL_PARAMS_COMMAND
4	NVGENNS	NVGENNS_COMMAND
5	NVGENS	NVGENS_COMMAND
6	PARAMS	PARAMS_COMMAND
7	PARAMS_ALIST	PARAMS_ALIST_COMMAND
8	PARAMS_KEYONLY	PARAMS_KEYONLY_COMMAND
9	PARAMS_KEYONLY_MM	PARAMS_KEYONLY_MM_COMMAND
10	FUNC	FUNC_COMMAND

The difference between the new macros and the deprecated macros is the addition of a *flags* field as the last parameter. This field gives the parser an indication of the capabilities of the action function that is called. The values for this flag are defined in the [EOL_* flags](#), which are described in the *Cisco IOS API Reference*.

One additional change is that the macros no longer provide their own extern definition of the action function. Consequently, the action functions must be defined before the macro is used. This was done because the function can now take on one of the following two types:

```
void (*old_func_type)(parseinfo *)
```

Or:

```
CONFIG_RC (*new_func_type)(parseinfo *)
```

The CONFIG_RC return values can be used instead of setting a PRC block within the action function if the EOL_USE_RETURN_CODE flag is added to the flags field of the new macros.

27.5.2 PRC Phase III and Action Function Return Codes

PRC Phase III improves the parser return code implementation in IOS by removing the various local definitions (such as sub exit codes) and adding consistent definitions for use by all branches. (For more detailed information on PRC Phase III, see EDCS-415801.)

PRC Phase III includes:

- Error Codes and Failure Types—Command failures are set to specific values and failures are set as temporary or permanent.
- Change Types and Modes—Data describes the changes made by successful commands and data describes the timing of any changes due to successful commands.
- An AF (Action Function) Facility—Allows action functions to declare that they return codes directly and to have the parser fill in the PRC block automatically.

PRC Phase III deletes the following information that is no longer needed:

- PRC Failure Severities
- Sub Exit Codes
- Component IDs

Note PRC Phase I and II APIs are deprecated and will be removed.

27.5.2.1 Error Codes

The error code enumerations are maintained by CANA and are available to applications for their own use. Action functions must have error codes of type `prc_error_code_t` that consistently describe the failure. For example:

- `PRC_INVALID`—Indicates an as yet unset error code. This is used by the infrastructure to initialize the error code.
- `PRC_SUCCESS`—Indicates a successful command (PRC was successfully returned).
- `PRC_GENERAL_FAILURE`—Indicates a general command failure has occurred.

Here is the current list of PRC error codes:

- `PRC_EAGAIN`
- `PRC_ENOMEM`
- `PRC_INVALID`
- `PRC_SUCCESS`
- `PRC_GEN_FAILURE`
- `PRC_BAD_SUBCOMMAND`
- `PRC_PARSE_ERROR_AMBIG`
- `PRC_PARSE_ERROR_INVALID`
- `PRC_PARSE_ERROR_LOCKED`
- `PRC_PARSE_ERROR_NOERR`
- `PRC_PARSE_ERROR_NOMATCH`
- `PRC_PARSE_ERROR_NOMATCH_ALIAS`
- `PRC_PARSE_ERROR_NOMEM`
- `PRC_PARSE_ERROR_REMOTE_FAILURE`
- `PRC_PARSE_ERROR_UNKNOWN`
- `PRC_PARSE_ERROR_CFG_DENIED`
- `PRC_PARSE_ERROR_NOAUTH`
- `PRC_UNINITIALIZED`

- PRC_NOT_ENABLED
- PRC_ALREADY_ENABLED
- PRC_NO_MEMORY
- PRC_BUSY
- PRC_INVALID_PARAMETER
- PRC_MSG_SEND_ERROR
- PRC_CHECKPOINT_FAILURE
- PRC_IO_FAILURE
- PRC_UNSUPPORTED_FEATURE
- PRC_INITIALIZATION_FAILED
- PRC_INVALID_CMD_SEQ
- PRC_PARSE_ERROR_STBY_NOMATCH

The meanings of the error code constants have not been officially documented but can be inferred by the user.

27.5.2.2 Failure Types

The failure type `prc_failure_type_t` enumerations are used to give information on whether the failure is temporary or permanent. For example:

- PRC_FAILURE_RETRY—Failure is to be considered temporary (for example, a resource is temporarily unavailable, such as a `malloc` failure).
- PRC_FAILURE_PERMANENT—Failure is to be considered permanent.

27.5.2.3 Change Types

The change type `prc_change_type_t` enumerations are used to give information on the type, if any, of the changes that were made by the successful command. For example:

- PRC_CONFIG_NO_CHANGE—No changes have occurred.
- PRC_CONFIG_CHANGE—Changes have occurred.
- PRC_CONFIG_CHANGE_NO_NVGEN—Changes have occurred, but will not be shown as part of nvgen output.

27.5.2.4 Change Modes

The change mode `prc_change_mode_t` enumerations are used to give information on the timing of the changes, because some configuration commands do not take effect immediately. For example:

- PRC_IMMEDIATE—All changes due to the successful command take effect immediately. No changes are queued pending an event of any sort.
- PRC_REBOOT—Commands where any effect will not take place until after a reboot. Note that for PRCs to convey information reliably it is required that all CLI commands operate in a deterministic manner. That is, by the time the action function associated with a command returns, any possible effects due to that command are expected to have taken place. If the state of the router upon completion of the action function is indeterminate, then this is to be considered a serious bug. This may include non-blocking commands that simply queue an event and then

return, the intent being that the event processing code will update the configuration as appropriate. In this case, a follow-up command may fail because it required the configuration changes due to the preceding command to have taken place.

- `PRC_DEFERRED`—Deprecated. Please do not use this value.
- `PRC_TIMING_UNKNOWN`—This value is for backwards compatibility only. If the PRC is set using one of the older APIs, which had no notion of change timing, then `prc_get_change_mode` will return `PRC_TIMING_UNKNOWN`.

27.5.2.5 PRC Phase III API

The following API functions were added (or modified in the case of `parse_cmd()`) to support PRC Phase III (see the *Cisco IOS API Reference* for details on these functions):

- 1 `parse_cmd()` (modified)
- 2 `prc_success()`
- 3 `prc_failure()`
- 4 `prc_fixup()`
- 5 `prc_map_errno()`
- 6 `prc_get_block_ptr()`
- 7 `prc_get_error_code()`
- 8 `prc_get_change_mode()`
- 9 `prc_get_change_type()`
- 10 `prc_get_is_temp_failure()`
- 11 `prc_ha_get_sync_enable()`
- 12 `prc_ha_set_sync_enable()`

27.5.2.6 New PRC Phase III AF (Action Function) Return Codes

In addition to the PRC API functions, a provision has been added to allow action functions to return codes from a small set of enums and have the parser fill out the PRC III block itself. This is a considerably simpler interface than the full PRC III API, and will simplify the code in situations where it can be used.

In order to use the new return code from action functions, a flag must be set on one of the new end of line macros to tell the parser that the action function is returning a value. The `EOL_PRC_SUPPORTED` flag is removed, and the `EOL_USE_RETURN_CODE` flag is added. This tells the parser that the action function included in that macro will return a value of type `CONFIG_RC`. That type is an enum, and it is defined in the `/vob/ios.comp/parser/include/config_base.h` file.

The possible values are:

- `CONFIG_RC_OK` - No error, and a change occurred.
- `CONFIG_RC_OK_NO_CHANGE` - No error, but also no change in config.
- `CONFIG_RC_OK_NO_NVGEN` - No error, but not saved to nvgen.
- `CONFIG_RC_FAIL_PERMANENT` - Permanent error occurred.
- `CONFIG_RC_FAIL_TEMPORARY` - Temporary error occurred, retry.

In addition, when using either CONFIG_RC_FAIL_PERMANENT or CONFIG_RC_FAIL_TEMPORARY, you can OR in any of the values from /vob/ios.comp/parser/include/enum/error_code.prc. This gives more information about what the real problem was. Only the bottom 8 bits are available for this. Other bits will cause the return code not to be recognized as such.

27.5.2.7 Sample Using PRC Phase III

For example, complete the following steps for PRC III:

Step 1 Get the prc block from the csb:

```
prc_t prc = prc_get_block_ptr(csb);
```

Step 2 Set PRC Success codes. For success *without* change to the running-config:

```
prc_success(prc, PRC_CONFIG_NO_CHANGE, PRC_IMMEDIATE);
```

For success *with* change to the running-config:

```
prc_success(prc, PRC_CONFIG_CHANGE, PRC_IMMEDIATE);
```

Step 3 Set PRC failure codes. Use the prc_failure() API. For any memory failure errors, use:

```
prc_failure (prc, PRC_NO_MEMORY, PRC_FAILURE_RETRY);
```

Step 4 Use the new EOLS_COMMAND macro (change EOLS to EOLS_COMMAND), where the last argument indicates if PRC is supported or not:

```
EOLS_COMMAND (conf_snmp_tftp_eol, snmp_ip_command, SNMP_TFTP_LIST,
EOL_PRC_SUPPORTED);
```

Note This macro will leave a “signature” in the sun file and the “signature” will help us track PRC compliance for TL9K, Sarbanes-Oxley.

The new PRC Phase III code is in bold font in this sample using prc_success and prc_failure:

```
void hostname_command (parseinfo *csb)
{
    char *new_hostname;
prc_t prc = prc_get_block_ptr(csb);

    if(csb->nvgen) {
        if(reg_invoke_general_alt_cmd(GENERAL_CMD_ALT_HOSTNAME_TYPE_NVGEN,
                                      csb) == FALSE) {
            if ((platform_get_value(PLATFORM_VALUE_SUPPORTS_DOCSIS11)) &&
                (NULL != strchr(hostname, ' '))) {
                nv_write((boolean)hostname, "!\\n%s \"%s\"\n!", csb->nv_command, hostname);
                return;
            }
            nv_write((boolean)hostname, "!\\n%s %s\\n!", csb->nv_command, hostname);
            return;
        } else {
            /*
             * the above reg_invoke has taken care of things
             */
            return;
        }
    }
}
```

```

        }

    }

    if(reg_invoke_general_alt_cmd(GENERAL_CMD_ALT_HOSTNAME_TYPE_ACCEPT,
                                  csb) == FALSE) {

        /*
         * do the normal thing
         * else the above reg_invoke would've taken care of things for us
         */

        new_hostname = GETOBJ(string,1);
        if ((NULL == new_hostname) || ('\0' == new_hostname[0])) {
            csb->sense = FALSE; /* treat as "" as "no hostname" command */
        }

        if (csb->sense) {
            if (!(platform_get_value(PLATFORM_VALUE_SUPPORTS_DOCSIS11))) {
                if (NULL != strchr(new_hostname, ' ')) {
                    /*
                     * Space allowed only in DOCSIS11 complaint platforms
                     */
                    printf("%% Spaces are not allowed in hostname \n");
                    prc_failure(prc, PRC_HOSTNAME_FAILED, PRC_FAILURE_PERMANENT);
                    return;
                }
            }
            if (OK != hostname_verify(new_hostname)) {
                printf("\n%% Hostname contains one or more "
                      "illegal characters. \n");
            }
            set_hostname(GETOBJ(string,1));
            gotname = TRUE;
        } else {
            set_hostname(platform_get_string(PLATFORM_STRING_DEFAULT_HOSTNAME));
            gotname = FALSE;
        }

        /*
         * Here we assume that the call to reg_invoke_hostname_changed always
         * succeeds. If that isn't the case then reg_invoke_hostname_changed
         * must be altered to enable it to convey error information, which can
         * then be checked on it's return.
         */
        reg_invoke_hostname_changed(gotname);
        prc_success(prc, PRC_CONFIG_CHANGE, PRC_IMMEDIATE);
    } else {

        /*
         * The call to reg_invoke_general_alt_cmd may already have set the PRC.
         * If not then we should set it now.
         */
        prc_error_code_t error = prc_get_error_code(prc);
        if (error != PRC_INVALID && error != PRC_SUCCESS) {
            prc_failure(prc, PRC_HOSTNAME_FAILED, PRC_FAILURE_PERMANENT);
        }
    }
}

```

Here is another example of code before the PRC III changes:

```
/*
 * snmp-server trap-timeout <seconds>
 * no snmp-server trap-timeout [<seconds>]
 */
EOLS(conf_snmp_timeout_eols, snmp_service,
      SNMP_TIMEOUT);

void snmp_service (parseinfo *csb)
{
...
    case SNMP_TIMEOUT:
        if (csb->sense) {
            snmp_def_timeout = GETOBJ(int,1) * ONESEC;
        } else {
            snmp_def_timeout = SNMP_DEF_TIMEOUT;
        }
        break;
...
}
```

Here is an example of the code after the PRC III changes (shown in bold font):

```
/*
 * snmp-server trap-timeout <seconds>
 * no snmp-server trap-timeout [<seconds>]
 */
EOLS_COMMAND (conf_snmp_timeout_eols, snmp_service,
                  SNMP_TIMEOUT, EOL_PRC_SUPPORTED);

void snmp_service (parseinfo *csb)
{
...
    prc_error_code_t error_code = PRC_GEN_FAILURE;
    prc_change_type_t change_type = PRC_CONFIG_CHANGE;
    prc_t prc = prc_get_block_ptr(csb);
    ulong new_snmp_def_timeout;

...
    case SNMP_TIMEOUT:
        if (csb->sense) {
            new_snmp_def_timeout = GETOBJ(int,1) * ONESEC;
        } else {
            new_snmp_def_timeout = SNMP_DEF_TIMEOUT;
        }
        if (new_snmp_def_timeout != snmp_def_timeout) {
            snmp_def_timeout = new_snmp_def_timeout;
        } else {
            change_type = PRC_CONFIG_NO_CHANGE;
        }
        break;
...
    if (sc_status != SC_PASS) {
        error_code = translate_sc_rc_to_prc(sc_status);
        prc_failure(prc, error_code, PRC_FAILURE_PERMANENT);
    } else {
        prc_success(prc, change_type, PRC_IMMEDIATE);
    }
}
```

```
...
```

Here is an example that uses AF Return Codes. Note that if you are going to return an error code, you *must* set the PRC_USE_RETURN_CODE flag in the macro.

The parse chain ends with:

```
extern CONFIG_RC log_config_command(parseinfo *);  
EOLNS_COMMAND (log_conf_eol, log_config_command, EOL_USE_RETURN_CODE)
```

and then, the action function would look like this:

```
CONFIG_RC log_config_command (parseinfo *csb)  
{  
    if (!csb->sense) {  
        /* Disable the config logger and reset to defaults */  
        if (is_config_log_enabled() == csb->sense) {  
            return CONFIG_RC_OK_NO_CHANGE;  
        } else {  
            config_log_disable();  
            return CONFIG_RC_OK;  
        }  
    }  
  
    if (csb->nvgem) {  
        if (set_mode_byname(&csb->mode,  
                           csb->new_mode_name, MODE_SILENT)) {  
            if (is_config_log_enabled() ||  
                is_parser_config_log_persist_reload_set()) {  
                nv_write(TRUE, "log config");  
            }  
        }  
        return CONFIG_RC_OK_NO_CHANGE;  
    } else {  
        if (set_mode_byname(&csb->mode,  
                           csb->new_mode_name, MODE_VERBOSE)) {  
            if (csb->flags & CONFIG_HTTP) {  
                char *new_uri = string_getnext();  
  
                sprintf(new_uri, STRING_BUF_CHARS,  
                        "%s", get_mode_name(csb->mode));  
                reg_invoke_http_parse_url(csb->http_state, csb->mode,  
                                           new_uri);  
            }  
            return CONFIG_RC_OK;  
        } else {  
            return CONFIG_RC_FAIL_PERMANENT | PRC_CHECKPOINT_FAILURE;  
        }  
    }  
    return CONFIG_RC_FAIL_PERMANENT;  
}
```

27.5.2.8 Migration from PRC Phase I and II

The existing implementations of the PRC APIs must be rewritten using the new Phase III APIs. Existing consumers of the older APIs will automatically be moved to the new implementations transparently (that is, the old APIs can be called, but they will transparently invoke the PRC III code).

implementation). However, the new rewritten code should be moved away from the older APIs so that the older APIs can be removed from the code base. This migration must be done on a per-component basis and is the responsibility of the component owners.

Here is a real example for the **rtr key-chain <name of key-chain>** command:

In the `/vob/ios/sys/rtt` directory, there is a file `cfg_rttmon.h`. In this file, a search for the word “key-chain” finds this `EOLS` call:

```
EOLS (config_rtr_keychain_eol, rttmon_config_base_command,
      RTR_CFG_KEYCHAIN_CMD);
```

Now a search in the directory for the `rttmon_config_base_command()` function finds that the function is present in the `rtt_cfg.c` file. In that function, the following old code is found and the new PRC Phase III code is in bold font to show the migration:

```
prc_t prc = prc_get_block_ptr(csb);

if (csb->nvggen) {
    ...
}

switch (csb->which) {
...
case RTR_CFG_KEYCHAIN_CMD:
    if (csb->sense) {
        if (!rtt_keychain_name_set(GETOBJ(string,1))) {
            printf("\n%%RTR: key-chain name too long");
            /* error is detected here */
            prc_failure(prc, PRC_GEN_FAILURE, PRC_FAILURE_PERMANENT);
        } else {
            prc_success(prc, PRC_CONFIG_CHANGE, PRC_IMMEDIATE);
        }
    } else {
        rtt_keychain_name_init();
        /* keychain is removed */
        prc_success(prc, PRC_CONFIG_CHANGE, PRC_IMMEDIATE);
    }
    break;
...
}
```

27.5.2.9 Example with Verification

This section describes how PRC III compliance is completed.

You need to test the behavior of existing and new commands, turn on PRC exposure with the **terminal prc { expose | hide }** command, and check that the following is true:

- 1** Are the PRC values exposed for your command?
- 2** Does re-applying the command result in “No Change”?
- 3** Are the correct error codes set for command failures?

An example of these steps is shown here (what you check in the output is shown in bold text):

Step 1 Enable PRC printing:

```
RTR1#terminal prc expose
!PRC[success, change-type=PRC_CONFIG_CHANGE_NO_NVGEN,
change-mode=PRC_IMMEDIATE, ha-sync-enable=false][command=terminal prc expose]
```

Step 2 Enter config mode:

```
RTR1#config terminal
Enter configuration commands, one per line. End with CNTL/Z.
!PRC[invalid][command=conf t]
RTR1(config)#
```

Step 3 Add the command:

```
RTR1(config)#snmp-server host 1.2.3.4 version 3 auth RamK
```

Step 4 Check for PRC success:

```
!PRC[success, change-type=PRC_CONFIG_CHANGE, change-mode=PRC_IMMEDIATE,
ha-sync-enable=true][command=snmp-server host 1.2.3.4 version 3 auth RamK]
RTR1(config)#
```

Step 5 Enter an incorrect command and check for PRC failure:

```
RTR1(config)#snmp-server host 1.2.3.4 vrf NOT_THERE version 3 auth RamK
% VRF name NOT_THERE does not exist
!PRC[failure, error-code=3005, failure-type=PRC_FAILURE_PERMANENT,
ha-sync-enable=false][command=snmp-server host 1.2.3.4 vrf NOT_THERE version
3 auth RamK]
```

27.5.3 Configuration Defaults Exposure

Configuration Defaults Exposure was developed to expose the complete configuration details that are suppressed by component-specified action functions to keep NVGEN output short.

For more detailed information about Configuration Defaults Exposure functionality, see EDCS-416532.

27.5.3.1 Configuration Defaults Exposure API

The following API macros were added to support Configuration Defaults Exposure (see the *Cisco IOS API Reference* for details on these macros):

- 1 [nv_add_check_default\(\)](#)
- 2 [nv_write_check_default\(\)](#)

27.5.3.2 Sample Using Configuration Defaults Exposure

You will need to complete the following steps:

Step 1 Examine the current nvgen code. The typical default logic looks like:

```
if (max_entries != PARSER_CONFIG_LOG_DEFAULT_ENTRIES)
{
    nv_write(TRUE, " %s %u", csb->nv_command, max_entries);
}
```

Step 2 Change the nvgen code; replace the `nv_write()` function with the `nv_write_check_default()` function. The new default logic looks like:

```
if (max_entries != PARSER_CONFIG_LOG_DEFAULT_ENTRIES)
{
    nv_write_check_default(csb, TRUE,
```

```

        max_entries != PARSER_CONFIG_LOG_DEFAULT_ENTRIES,
        " %s %u", csb->nv_command, max_entries);
    }
}

```

The new Configuration Defaults Exposure code added by the component developers is in bold font in this sample using the `cdp_command()` action function:

```

void cdp_command (parseinfo *csb)
{
    idbtype *swidb;
    ipaddrtype ipaddress;
    prctype prc_str;
    prc_get_rc_default(&prc_str);
    prc_str.comp_id = CDP_COMP_ID;

    if (csb->nvg) {
        switch (csb->which) {
            case CDP_RUNNING:
                if (!PLATFORM_WANTS_CDP) {
                    /* Platform wants cdp configured by default */
                    nv_write_check_default(csb, TRUE,
                        !cdp_running,
                        !cdp_running ? "no %s", csb->nv_command);
                } else {
                    /* Platform does NOT want cdp configured by default */
                    nv_write_check_default(csb, TRUE,
                        cdp_running,
                        cdp_running ? "no %s" : "%s", csb->nv_command);
                }
                break;
            case CDP_CONFIG_TIMER:
                nv_write_check_default (csb, TRUE,
                    xmit_frequency != CDP_DEF_XMIT_INTERVAL,
                    "%s %d", csb->nv_command, cdp_xmit_frequency);
                break;
            case CDP_CONFIG_HOLDTIME:
                nv_write_check_default (csb, TRUE,
                    cdp_holdtime != CDP_DEF_TTL,
                    "%s %d", csb->nv_command, cdp_holdtime);
                break;
            case CDP_SOURCE_INT:
                if (cdp_source_int_flag && cdp_source_int_idb) {
                    nv_write_check_default (csb,
                        cdp_source_int_flag && cdp_source_int_idb,
                        TRUE, "%s %s", csb->nv_command,
                        cdp_source_int_idb->namestring);
                }
                break;
            case CDP_ADVERTISE:
                nv_write_check_default (csb, TRUE,
                    cdp_advertise_v2 != CDP_DEFAULT_ENABLE_V2,
                    "%s %s", cdp_advertise_v2 == FALSE ? "no" : "",
                    csb->nv_command);
                break;
            case CDP_LOG_DUP_MISMATCH:
                nv_write_check_default (csb, TRUE,
                    !cdp_log_duplex_mismatch,
                    !cdp_log_duplex_mismatch : "no %s", "%s",
                    csb->nv_command);
                break;
        }
    }
}

```

```

        default:
            bad_parser_subcommand(csb, csb->which);
            break;
    }
    return;
}
. . .

```

27.5.3.3 Example with Verification

Here is an example that shows how Configuration Defaults Exposure compliance is completed (the following steps and what you check in the output is shown in bold text).

Step 1 Enter config mode:

```
7603-2#conf terminal
Enter configuration commands, one per line. End with CNTL/Z.
```

Step 2 Add the command with the default value:

```
7603-2(config)#snmp-server inform timeout 3
7603-2(config)#end
```

Step 3 Check show running-config:

```
7603-2#show running-config | include timeout
7603-2#
```

Step 4 Check show running-config all:

```
7603-2#show running-config all | include timeout
snmp-server inform timeout 3
7603-2#
```

Step 5 Enter config mode and check the command with a nondefault value:

```
7603-2#conf terminal
Enter configuration commands, one per line. End with CNTL/Z.
7603-2(config)#snmp-server inform timeout 15
7603-2(config)#end
```

Step 6 Check show run.

```
7603-2#show running-config | include timeout
snmp-server inform timeout 15
```

27.5.4 Rollback Component Conformance

Rollback Component Conformance requires that components institute rollback compliance testing to validate that all IOS configuration commands can be rolled back (i.e. can be removed from the running configuration). Any exceptions found must be documented and registered with the config rollback API, and the problem must be addressed if possible. This section is based on information in EDCS-438784.

27.5.4.1 The Config Rollback API

The following API functions are provided to support Rollback Component Conformance (see the *Cisco IOS API Reference* for details on these functions):

- 1 `reg_invoke_add_ignore_command()` - Use this registry function for commands that should be ignored because they do not need to roll back (such as, the **version** and **timestamp** commands). For example:

```
reg_invoke_add_ignore_command("no ip address");
```

- 2 `reg_invoke_add_special_negation_command()` - Use this registry function for commands that do not use the typical **no** command to be negated (such as, the **transport input none** command, which is negated with the **default transport input none** command). For example:

```
reg_invoke_add_special_negation_command ("transport preferred none", "default transport preferred");
```

- 3 `reg_invoke_add_undo_command()` - Use this registry function for commands that use the typical **no** command to be undone, but also need the first *n* number of words of the command to be undone. For example, register the **banner motd** command, which is undone with the **no banner motd** command. To keep all words, use the **KEEP_ALL_WORDS** constant:

```
reg_invoke_add_undo_command ("banner", 2);
```

- 4 `reg_invoke_add_use_parent_to_undo_command()` - Use this registry function for commands that cannot be undone using the typical **no** command, and must be undone by undoing the parent. For example, the parent **ip vrf vrf_instance_name** command must be used with the **rd VPN_Route_Distinguisher** command to undo the **rd** command, so register the command:

```
reg_invoke_add_use_parent_to_undo_command ("rd", "ip vrf");
```

Users of the Config Rollback API functions must include:

```
#include "ui/rollback_registry.h"
```

27.5.4.2 Example with Verification

Here is an example that shows how rollback conformance is completed with the following steps shown in bold text.

Step 1 Save the configuration:

```
7603-2#copy running-config disk0:runcfg-working
Destination filename [running-config]?
6462 bytes copied in 1.432 secs (4513 bytes/sec)
```

Step 2 Add the command in config mode:

```
7603-2(config)#snmp-server ifindex persist
7603-2(config)#end
```

Step 3 Verify the changes:

```
7603-2#show archive config differences disk0:runcfg-working
Contextual Config Diffs:
- snmp-server ifindex persist
7603-2#
7603-2#
7603-2#show running-config | include snmp-server ifindex
```

```
snmp-server ifindex persist
```

Step 4 Roll back the changes:

```
7603-2#config replace disk0:runcfg-working force  
Total number of passes: 1  
Rollback Done
```

Step 5 Verify the rollback:

```
7603-2#show running-config | include snmp-server ifindex
```

Note Nothing shows up.

27.5.5 Support for Syntax Checking

Syntax checking was developed because customers want a “Two Phased Commit” for configuration changes:

- Phase 1: Check syntax. *Do not apply*.
- Phase 2: Apply at the appropriate time.

Syntax checking:

- 1 Checks **config** commands without applying them (syntax checking does not apply to **exec** commands).
- 2 Provides a new syntax check mode that *looks like* config mode.
- 3 Allows customers can “simulate” configuration actions without effecting changes.

27.5.5.1 Issues with Syntax Checking

Issues with syntax checking stem from the fact that we are retro-fitting code that is over 10 years old. Some submodes create configuration changes during syntax check. For example, if you enter **interface loopback0** in syntax check mode and another user telnets into the router and enters a **show running config** command, they will see “interface loopback0” in the running config.

Do your best to avoid configuration changes, but if you cannot avoid the config changes, you should:

- Register a cleanup function as described in this section, to clean up your submode data when checking is complete.
- Document these exceptions to make customers are aware of them.
- Ask customers to “lock” the config before doing a syntax check. This will avoid config changes issued by another user, while the customer is syntax checking. The locking mechanism for configs is already available on the S and T trains. You have the ability to lock only the **config** commands from other users, or to lock **config** and **show** commands.

27.5.5.2 How to Support Syntax Checking

Syntax-checking mode is exactly the same as normal configuration mode, except that the component action function is not called. The main issue with this is that submodes are entered using **set_mode_byname()** in the action function of some commands. Also, modes are exited by the action function of exit commands that are part of the chain. So, in these two cases, the action function must be called, but must be able to determine that it is being called while syntax checking.

To overcome this problem, the component developer must use one of the new macros from Table 27-5, and pass a flag to tell the parser that it should be called during syntax-checking mode.

When the parser is in syntax-checking mode, the component action function will only be called if one of the following flags is present:

- `EOL_MODE_ENTRY`—Use for an AF that enters a mode.
- `EOL_MODE_EXIT`—Use for an AF that exits a mode.
- `EOL_SYNTAX_CHECK`—Use for any other situations where calling an AF in syntax-checking mode is required; for example, the `FUNC_COMMAND` macro.

These are added to the `flags` field of the `EOLS_COMMAND`, `EOLNS_COMMAND`, `PARAMS_COMMAND`, `PARAMS_ALIST_COMMAND`, `PARAMS_KEYONLY_COMMAND`, or `PARAMS_KEYONLY_MM_COMMAND` macros. Within the action function, the `csb->parse_cmd_syntax` should be used to determine if the parser is checking syntax only, and so must not modify existing configuration, or must register callbacks for later deletion of configuration.

An important task that must be done to support syntax checking is to check every `ASSERT`, `TEST`, and `IFELSE` macro that is in a parser sub-mode in your component to see if it tests a value that was set by the mode-entering command. If so, you must modify the test so that it has the same effect when `csb->parse_cmd_syntax` is `TRUE`, or modify the action function that enters the submode to simulate configuration mode in such a way that these macros are satisfied, and can continue to check configuration. The aim is to make syntax-checking mode accept exactly the same commands as if it were actually configuring, but to not accept any other commands. You can register a callback function to clean up any added configuration using `parser_set_syntax_cleanup()`.

In addition to the above changes, any macro that provides an exit to a submode must also be run in syntax-checking mode. Replace the `EOLS` and `EOLNS` macros for sub-mode exit functions with `EOLS_COMMAND` and `EOLNS_COMMAND` macros, and add the flag `EOL_MODE_EXIT` to the `flags` field. This sets proper navigation through the various sub-modes even though nothing is being configured during syntax checking. This applies to any functionality that needs to be run in both configuration and syntax-checking modes.

For example:

```
EOLS_COMMAND(ci_FR_dlcippp_eol, frame_relay_command,  
FR_DLCL, EOL_MODE_EXIT);
```

27.5.5.2.1 CLI for Syntax Checking

The command to place the router in syntax-checking mode is a hidden command (service internal only) that can be issued from the exec mode. This command will be unhidden when enough support exists for it. An example of the command is shown here:

```
Router#config check syntax  
Enter the configuration commands, one per line. End with CNTL/Z.  
Router(syntax)#
```

From the `Router(syntax)#` prompt, enter all commands as if configuring the router to syntax check them. You will remain in syntax-checking mode until you return to the exec mode.

The additional `config check syntax file` exec command has been created to allow you to check the syntax of a file for all file types that can be copied to the running-config, as shown here:

```
Router#config check syntax unix:config_net  
Destination filename [running-config]?  
garbage command  
^
```

```
% Invalid input detected at '^' marker.

193 bytes copied in 0.64 secs
Router#
```

27.5.5.3 Examples with Verification

The examples in this subsection are provided to show you how to change your command to be compliant with syntax checking.

The first example is a non-submode command that uses the default syntax check handler function. The old macro used is EOLS and the **ip http client password <password>** command is used for the http client:

```
EOLS (cfg_httpc_auth_password_eol,
      http_client_cfg_command,
      PARSE_HTTP_AUTH_PASSWD);

STRING (cfg_http_client_password_passwd_enc,
        cfg_httpc_auth_password_eol, no_alt,
        OBJ(string,1), "The HIDDEN client password
        string");

TEXT (cfg_http_client_password_passwd,
      cfg_httpc_auth_password_eol, no_alt,
      OBJ(string,1), "The UNENCRYPTED (cleartext)
      client password");

TEST_MULTIPLE_FUNCS(cfg_http_client_password_test,
                    cfg_http_client_password_passwd,
                    no_alt, NONE);
```

In this example, replace the EOLS macro with EOLS_COMMAND:

```
EOLS_COMMAND(cfg_httpc_auth_password_eol,
              http_client_cfg_command,
              PARSE_HTTP_AUTH_PASSWD, EOL_PRC_SUPPORTED |
              EOL_MODE_ENTRY);

STRING (cfg_http_client_password_passwd_enc,
        cfg_httpc_auth_password_eol, no_alt, OBJ(string,1),
        "The HIDDEN client password string");

TEXT (cfg_http_client_password_passwd,
      cfg_httpc_auth_password_eol, no_alt, OBJ(string,1),
      "The UNENCRYPTED (cleartext)
      client password");

TEST_MULTIPLE_FUNCS(cfg_http_client_password_test,
                    cfg_http_client_password_passwd,
                    no_alt, NONE);
```

The **http_client_cfg_command()** function must then check whether **csb->parse_cmd_syntax == TRUE**. If so, it must only enter the mode, and not do any of the other configuration it would normally do.

After you make your command syntax-checking compliant, you need to verify that syntax checking for your command works as expected. Developers can manually or automatically verify this. Manual verification is done by logging onto the router. Be sure to make sure that the command, when entered in syntax-checking mode, does not get configured and show up in the running configuration.

Here is an example of the **archive** command not being syntax check compliant. Therefore, the EOLS macro for the archive submode was not replaced by EOLS_SUBMODE.

Step 1 First, check for the command in the running-config:

```
gt3-7200-4#
gt3-7200-4#sh run | i archive
```

Step 2 Next, enter the syntax checking mode by entering the **config check syntax** command:

```
gt3-7200-4#conf check syntax
Enter configuration commands, one per line. End with CNTL/Z.
```

Step 3 Then, enter the **archive** command:

```
gt3-7200-4(syntax)#archive
gt3-7200-4(syntax)#end
gt3-7200-4#
```

As you can see, the **archive** submode was not entered, therefore syntax checking fails for the **archive** command. As mentioned before, commands should be able to be entered as if you were in configuration mode.

After making the **archive**, "log config" submode, and the **hidekeys** command syntax-checking compliant, you can check that syntax checking will pass:

Step 1 First, check for the commands in the running-config:

```
gt3-7200-4#
gt3-7200-4#sh run | i archive
gt3-7200-4#sh run | i hidekeys
```

Step 2 Next, enter the syntax-checking mode and enter the **archive** command and the "log config" submode:

```
gt3-7200-4#conf check syntax
Enter configuration commands, one per line. End with CNTL/Z.
gt3-7200-4(syntax)#archive
gt3-7200-4(syntax-archive)#log config
```

You can see the **archive** submode was properly entered.

Step 3 Then, enter the "log config" submode and **hidekeys** command:

```
gt3-7200-4(syntax-archive)#log config
gt3-7200-4(syntax-archive-log-cfg)#hidekeys
```

Step 4 Finally, verify that the **hidekeys** command was not configured by checking for it in the running config:

```
gt3-7200-4(syntax-archive-log-cfg)#end
gt3-7200-4#sh run | i hidekeys
gt3-7200-4#
```

You can also syntax check a whole config file. This doesn't require you to enter the syntax-checking mode or to enter commands one by one.

Step 1 First, check for the **hidekeys** command in the running config:

```
gt3-7200-4#more tftp://dirt/pratvenk/config_test_hidekeys
archive
log config
hidekeys
end
gt3-7200-4#sh run | i archive
gt3-7200-4#sh run | i hidekeys
```

Step 2 Next, syntax check the config file, by the entering the **config syntax check filename** command (where *filename* is **config_test_hidekeys**):

```
gt3-7200-4#conf check syntax tftp://dirt/pratvenk/config_test_hidekeys
Destination filename [running-config]?
Accessing tftp://dirt/pratvenk/config_test_hidekeys...
Loading pratvenk/config_test_hidekeys from 171.69.1.129 (via
FastEthernet0/0): !
[OK - 32 bytes]

32 bytes copied in 0.068 secs (471 bytes/sec)
%unable to unlock configuration mode
```

Step 3 Finally, verify that the **hidekeys** command was not applied in the running-config:

```
gt3-7200-4#sh run | i archive
gt3-7200-4#sh run | i hidekeys
gt3-7200-4#
```

27.5.6 Configuration Change Notification

Configuration Change Notification (CCN) was developed because customers want *exact change* notification details, such as:

- What changed?
- Who changed it?
- When?
- What is the PRC value?
- What does the new config look like?

The benefits of CCN are:

- Security audits
- Efficient change management
- Sarbanes-Oxley compliance

CCN includes the config logger that can be used to track and report configuration changes. The config logger supports two types of “contenttype”:

- 1 plain-text—With plain-text format, the config logger reports configuration changes only.
- 2 xml—With xml format, the config logger reports the configuration change details also (what, who, when, PRC, and incremental NVGEN results).

CCN includes “Config before-after” debugging that can be used to verify per command-based incremental NVGEN behaviors.

Here is an example of the CCN config logger:

Step 1 Prepare for using the config logger:

```
three#conf terminal  
Enter configuration commands, one per line. End with CNTL/Z.  
three(config)#archive  
three(config-archive)#log config
```

Step 2 Turn on the config logger:

```
three(config-archive-log-cfg)#logging enable
```

Step 3 Record the PRC results of the config commands:

```
three(config-archive-log-cfg)#record rc
```

Step 4 Record the incremental NVGEN results of the config commands:

```
three(config-archive-log-cfg)#record before-after  
three(config-archive-log-cfg)#end  
three#
```

Step 5 Configure a **snmp-server** command as the example:

```
three#conf t  
Enter configuration commands, one per line. End with CNTL/Z.  
three(config)#snmp-server inform timeout 5  
three(config)#end
```

Step 6 Show the config logger results (which shows here who=console, when=2006-03-07T22:52:01.379Z, what= snmp-server inform timeout 5, PRC=PRC_CONFIG_CHANGE, configuration before applying the command = end, and configuration after applying the command = snmp-server inform timeout 5, end):

```
three#sh archive log config all contenttype xml  
<?xml version="1.0" encoding="UTF-8"?>  
<configLoggerMsg version="1.0">  
<configChanged>  
<changeInfo>  
<sessionId>4</sessionId>  
<user>console</user>  
<async>  
<port>console</port>  
</async>  
<when>  
<absoluteTime>2006-03-07T22:52:01.379Z</absoluteTime>  
</when>  
</changeInfo>  
<changeItem>  
<context>  
</context>  
<enteredCommand>  
<cli>snmp-server inform timeout 5</cli>  
</enteredCommand>  
<prcResultType>  
<prcSuccess>  
<change>PRC_CONFIG_CHANGE</change>  
<changeMode>PRC_IMMEDIATE</changeMode>  
</prcSuccess>  
</prcResultType>  
<oldConfigState>  
<cli>
```

```
end
</cli>
</oldConfigState>
<newConfigState>
    <cli>
        snmp-server inform timeout 5
    end
</cli>
    </newConfigState>
    </changeItem>
    </configChanged>
</configLoggerMsg>
```

IOS features such as HA, ISSU, and Configure Change Notification Agent needed a mechanism to provide accurate information on the “before” and “after” configuration state, so each CLI parse chain must now generate its own “running-config” state to allow the above listed IOS features (HA, ISSU, CNS) to check for Config Change Notification support. Component developers need to instrument the configuration CLIs’ parse chains and action functions using the new EOL macros to generate the information and the IOS features to use Config Change Notification appropriately (for detailed information on Configuration Change Notification, see EDCS-416578).

The new EOL and PARAMS macros defined for the new “parse syntax check” parser feature have a EOL_NVGEN_SUPPORTED flag bit field that allows the IOS Parser to determine if an action function supports both configuration and nvgenning. Component CLIs must now define either NVGEN or new EOL/PARAMS macros.

Here is an example of CCN “Config before-after” debugging:

Step 1 Prepare for CCN “Config before-after” debugging:

```
cp-c7606-2#conf terminal
Enter configuration commands, one per line. End with CNTL/Z.
```

Step 2 Turn on service internal:

```
cp-c7606-2(config)#service internal
cp-c7606-2(config)#end
```

Step 3 Turn on **before-after** debug:

```
cp-c7606-2#debug parser config before-after
Parser config before and after per command debugging is on
```

Step 4 Configure the command being tested (**snmp-server inform timeout 5**):

```
cp-c7606-2#conf terminal
Enter configuration commands, one per line. End with CNTL/Z.
cp-c7606-2(config)#snmp-server inform timeout 5
Before:
cp-c7606-2(config)#
```

Step 5 Check the “nvgen” result before applying the command:

```
000033: *Mar 3 02:31:19: BEFORE:
end
:
```

Step 6 Check the “nvgen” result after applying the command:

```
000034: *Mar 3 02:31:19: AFTER:
snmp-server inform timeout 5
end
:
cp-c7606-2(config)#
```

Here is what developers need to check for incremental NVGEN:

Step 1 Examine existing the command behavior:

- (a) Check the action functions of NVGENS and EOL nodes, and make sure that they support the NVGEN request. For example:

```
void snmp_service (parseinfo *csb)
{
    ...
    if (csb->nvgen) {
        if (!snmp_serving) {
            return;
        }
        ...
    }
}
```

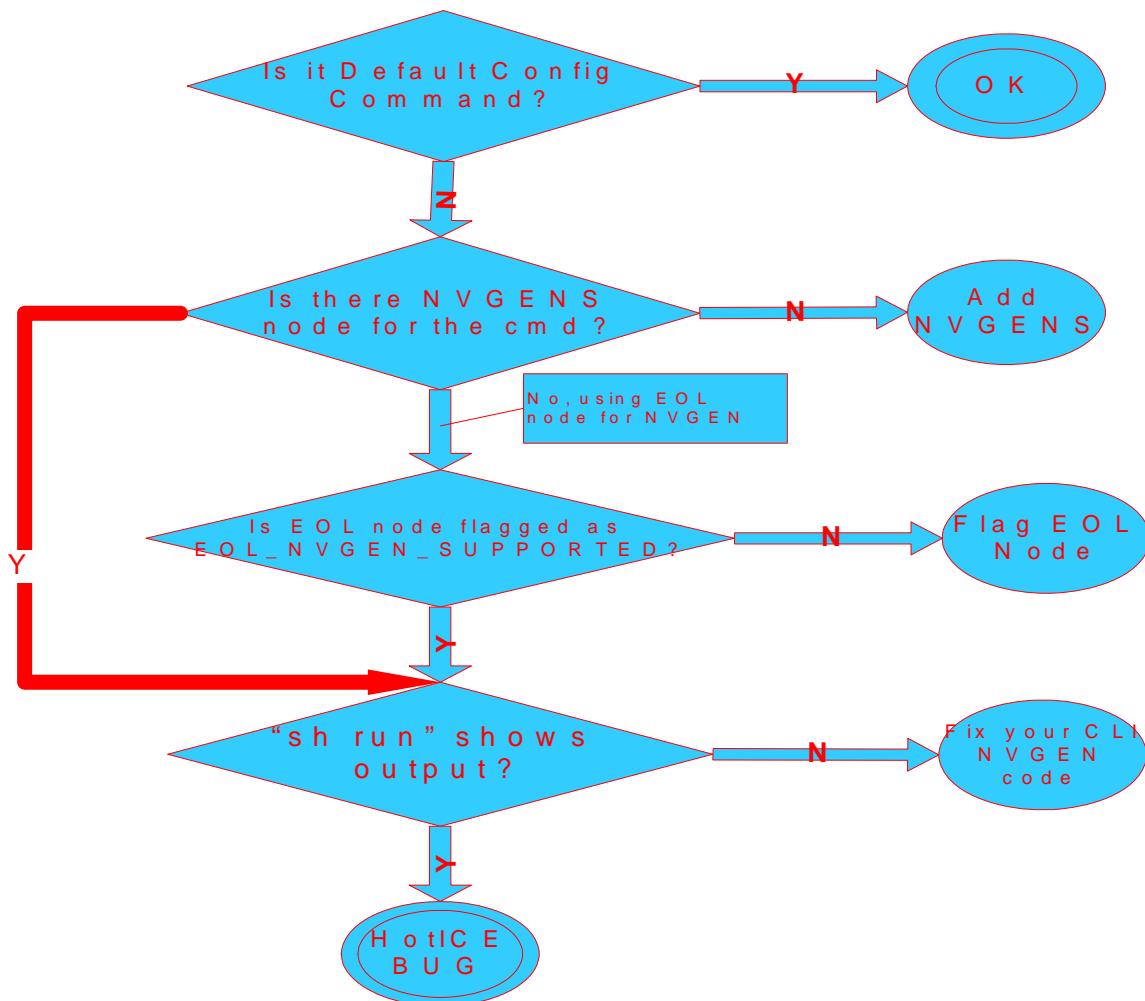
- (b) Turn on the config logger or before-after debugging, and check the command's runtime behavior.

Step 2 Classify exceptions:

- (a) Some commands generate *zero* output.

Add appropriate NVGEN code to the action function. Zero NVGEN could be caused by Parser Chain issues (missed NVGENS node, or NVGENS/EOLS nodes are not reachable during NVGEN traversal), CLI NVGEN issues (CLI action function does not handle NVGEN request well), or Hot ICE infrastructure defects. This is shown in Figure 27-5.

Figure 27-5 Zero NVGEN



- (b) Some commands regenerate the *entire* submode.
- Break up NVGEN logic among submode commands.
 - Make the CLI more modular. This is not always possible because of the “age of code”, complexity, or the authors of the code are no longer working at Cisco.
- (c) Some commands generate *voluminous* configs.

Work with Hot ICE team to “creatively” report changes. Use the `nv_write_before()` and `nv_write_after()` functions, and the `EOL_SKIP_INCREMENTAL_NVGEN` flag in the `EOLS_COMMAND`, to generate an appropriate before and after for the minimum portion of the normal nvgen total (for more information, see section 27.5.7).

Step 3 Use the new `EOLS_COMMAND` macro:

- If the `EOLS` node is reachable during NVGEN traversal, you need to flag it.
- The last argument indicates if CCN is supported or not:

```

EOLS_COMMAND (conf_snmp_tftp_eol, snmp_ip_command, SNMP_TFTP_LIST,
EOL_PRC_SUPPORTED);

```

- The EOLS_COMMAND macro will leave a “signature” in the sun file.
- The “signature” will help to track CCN compliance for TL9K and Sarbanes-Oxley.

For example, here is a sample of a code change (in bold font) to support CCN:

```
* cfg_snmp_inform.h: Configuration commands for SNMP informs
*
*****
*** 22,32 ***
* OBJ(int,3) = TRUE if timeout being configured
* OBJ(int,4) = timeout
* OBJ(int,5) = TRUE if pending being configured
* OBJ(int,6) = pending
*/
! EOLS  (conf_snmp_inform_eol, snmp_inform_service, SNMP_INFORM_KNOBS);

--- 22,34 ---
* OBJ(int,3) = TRUE if timeout being configured
* OBJ(int,4) = timeout
* OBJ(int,5) = TRUE if pending being configured
* OBJ(int,6) = pending
*/
! EOLS_COMMAND  (conf_snmp_inform_eol, snmp_inform_service,
!                 SNMP_INFORM_KNOBS, EOL_PRC_SUPPORTED);
```

27.5.7 Before After API

One problem with the INCREMENTAL_NVGEN implementation is that some action functions will generate a huge amount of data during nvgen. One example might be a big set of ACLs. A single call to an action function generates the entire list. Unfortunately, this means that the previous incremental nvgen scheme at the end of section 27.5.6 will generate the entire list that existed before as the “before” buffer, and the entire list that exists after the change as the “after” buffer. For 1000 ACLs, this means that a total of 1 million lines of config change information will be generated. This is obviously not very desirable.

In order to overcome this problem, developers can mark their action function node with a flag that indicates that no before/after incremental nvgen should be attempted for this action function. Also, two API functions exist that can be used to fill in the before and after buffers for config change notification.

The constant flag for EOLS_COMMAND, EOLNS_COMMAND, etc., is EOL_SKIP_INCREMENTAL_NVGEN.

The new APIs are `nv_write_before()` and `nv_write_after()` and to use them, follow these steps:

- Step 1** Set the flag in the EOL node that calls the action function.
- Step 2** Within the action function, before the config change, use `nv_write_before()` to output the prior configuration. In most cases, this will be NULL, in which case it can be skipped.
- Step 3** Make the configuration change.
- Step 4** Use `nv_write_after()` to output the changed configuration one more time. The output should reflect the changed configuration.

If the action function is called from a submode, the submode information must also be output, in order to provide context for the call.

The format for outputting the submode information should contain the mode entry command, separated by a slash \n (\n) for each submode.

27.5.7.1 Config Data that is Changed by Others (Side Effects)

Another use for nv_write_before/after function is when your configuration data can be changed by mechanisms other than the configuration CLI. In this case, you can create a “setter” function for your config data. For example, include a csb in your signature and call nv_write_before() and nv_write_after() including the CLI context required, as shown here:

```
nv_write_before(csb, TRUE, "int e1\\nip address %i", ip_address);
ip_address = new_ip_address;
nv_write_after(csb, TRUE, "int e1\\nip address %i", ip_address);
```

27.5.8 How to Instrument IOS Components for Hot ICE

The guidelines for instrumenting IOS components using the support provided by the infrastructure changes described in the previous sections for Hot ICE are provided in EDCS-569526.

27.5.9 Hot ICE Compliance Tooling

The Hot ICE compliance tooling helps IOS Components Development Engineers and Test Engineers automate the verification of the Hot ICE compliance of the IOS configuration commands during the unit testing and the regression testing phases. It allows you to build component or command-specific test cases and run the Hot ICE Compliance Test Suites against the given IOS image and the given platform, including the IOU (IOS On Unix) platform.

Hot ICE Compliance Tooling is available via both the web and UNIX command line (CLI) interfaces. The web GUI and CLI front-end user interfaces simplify the collection of the test inputs needed to build test cases from the user and also hide the complexity of managing/executing the backend Hot ICE compliance test suites on the test bed platforms. The results of the Hot ICE compliance test runs are emailed to the users and are also made available via the web interface for easy diagnostics and submission into CDETS. The test cases can be checked into the Hot ICE test repository for rerunning the tests by the same or a different user.

Refer to the *Hot ICE Compliance Tooling Users Guide* EDCS-515696 for information on how to use this tool. Or, simply visit <http://hotice.cisco.com>.

27.6 Linking 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.

27.6.1 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 `exec_disconnect.h` file, 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>
 */
EOLS    (exec_disconn_eol, disconnect_command, 0);

STRING  (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);
NUMBER  (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);

#undef ALTERNATE
#define ALTERNATE      exec_disconn

```

`exec_disable.h`

```

/*
 * disable
 */
EOLS    (exec_disable_endline, enable_command, CMD_DISABLE);
KEYWORD (exec_disable, exec_disable_endline, ALTERNATE,
          "disable", "Turn off privileged commands", PRIV_ROOT);

#undef ALTERNATE
#define ALTERNATE      exec_disable

```

27.7 Manipulating CSB Objects

27.7.1 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 27-6 lists the allowable object data types and numbers.

Table 27-6 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
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

27.7.2 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")
```

27.7.3 Important Notes on Data Variables

CSB variables are typically set during command parsing via appropriate parser macros and later retrieved using `GETOBJ` within the function processing for that command.

All `OBJ(xxx, nnn)` CSB members are cleared before command parsing begins. CSB members that are cleared are from `csb->txt1` (equated to `csb_start_data`) through `csb->end_of_data` (equated to `csb_end_data`).

The following paragraph provides more information about the CSB data variables (`csb_start_data` to `csb_end_data`) and their usage in a parse chain.

It is important to note that the parser carries these CSB data variables when it moves across the accept path in a parse chain. Basically, the parser stores the current data area of the CSB before moving on to the accept path because the data area of the CSB can change during the accept path movement. Once the parser is back to the alternate path, the data area of the CSB stored previously is restored. This way, the different CLIs can reuse the limited available data variables for their purposes.

Hence, CLIs should be written keeping this concept in mind. For example, a CLI should not expect the value of a data variable in the alternate path to be the one that was set in any of the previous accept paths.

Note that the parser does not use a storage/restoration mechanism for many of the CSB's data variables. Because this was a defect, it has been fixed in the new parser component. A new mechanism has also been added to detect a CSB variable that is used out of the defined scope.

27.7.3.1 Background of Parsing Flaw

Here is an example of a parsing flaw.

Sometimes, the empty output of a BGP regular expression is encountered using its abbreviation, **re** instead of the full word, **regexp**.

```
dist02.cas#sh ip bgp re ^701$  
dist02.cas#sh ip bgp reg ^701$
```

The parse subtree of show **ip bgp <options>** is shown in Figure 27-6:

Figure 27-6 Parse Subtree of show ip bgp <options>



For the command, **sh ip bgp re ^701\$**, **re** matches first with **replication** and sets the integer(int2) to TRUE. Later **re** matches with **regexp** also. Here the expected value of int2 is FALSE but it has been changed to TRUE in the previous accept path of **replication**. Since the action function checks for integer(int2) and performs some action based on int2 which is TRUE by mistake and its expected value is FALSE, the output is not correct. So, to solve such problems the parser must ensure that the correct CSB context is maintained across the accept and the alternate paths.

The parser already has some kind of the CSB's context store and restore operations. It will store the current data area of the CSB before moving on to the accept path. During the accept path the data of the CSB can change. Once the parser gets back to the alternate path, the data area of the CSB stored

previously will be restored. This is done using the functions `parser_copy_info_2_node()` and `parser_copy_node_2_info()`. Since all the data fields are not stored and restored in this function (like `int2`), some fields remain to hold the invalid values resulting in unexpected outputs.

Hence the fix is to modify the above two functions such that the entire CSB's data area will get stored and restored across accept and alternate paths.

Note that if “parser cache” is enabled, the above problem is only faced once because subsequent tries make use of the parser cache entry and not the entire parse tree.

The parser cache entry holds only the accept path and hence in this case, the path corresponding to replication is not present in the cache entry.

27.8 Ordering Commands

When NVGEN traverses the parse tree, it does an `accept-first` traversal of the tree. The submodes that are traversed depend on the implementation of the submode entry function. Sometimes the submode entry function simply writes out the NVGEN for the entire mode. If an NVGENS or NVGENNS node is encountered during this traversal, the `accept` subtree is not traversed.

The parse trees are assembled in “NVGEN order”, such that a standard traversal of the parse tree automatically calls components in the desired order for their configuration contribution. There is no separate “parse order” or secondary tree threading, since parsing speed issues are typically configuration-specific and are optimized by the parser cache and parser bookmarks. Ordering commands is about `nvgen` transversal order, which is only relevant for configuration. The `exec` mode enums above are not relevant to the `nvgen` traversal order; however, these are commonly used for linking to shared pre-existing system-wide keywords (`show`, `debug`, and `clear`).

The parse tree basically consists of a number of link points and parse nodes. The link points define the *order* that commands get written when you do a `show running` and also the *order* that they are parsed in the startup configuration (because the running configuration is typically copied to the startup configuration). This is important because one command may depend on another having already been parsed. You use a specific link point to force a specific ordering.

Note Of course, the end user could enter commands in a different order than you have them NVGEN'ed and, in general, that should be ok. There is usually little reason for not allowing commands to be entered in any order. There are some commands that do depend on order, but it is best for the end-user if you try to avoid this when possible.

`parser.h` contains the link point names. `chain.c` shows the order that these link points exist in the parse tree as it comes out of the box. You might have to look in here to see exactly where you want to link in your command, since you can't tell by looking at the enum in `parser.h`. There, the ordering has no meaning. But, typically, you'll want to add in your commands in one of very few places, so scrounging around in `chain.c` isn't usually necessary, even though you might find it fun.

The following list includes some of the most widely used link points. See `sys/parser/chain.c` for a complete list of link points.

- `PARSE_ADD_CFG_TOP_CMD`—A global configuration command that should appear BEFORE any interface command
- `PARSE_ADD_CFG_LAST_CMD`—A global configuration command that should appear AFTER all the interface commands
- `PARSE_ADD_SHOW_CMD`—A `show` command

- PARSE_ADD_DEBUG_CMD—A **debug** command
- PARSE_ADD_CLEAR_CMD—A **clear** command
- PARSE_ADD_EXEC_CMD—Any other exec command
- PARSE_ADD_CFG_INT_CMD—An interface configuration command and you don't have any dependencies between your command and other interface configuration commands

Maintaining the order of configuration commands grouped together, based on type, is important for readability and easy search. The following list includes some of the recommended configuration command order guidelines:

- 1 Configure the controlling commands before defining the supported interfaces.
- 2 You should code the hardware-dependent configurations to avoid the order dependency.
- 3 You can maintain the NVGEN order according to the code requirement and the amount of resource used. It should be independent of any orders as far as possible.
- 4 You can code the tied-up configurations so that they are independent of any command order.

For additional information or any clarifications regarding the grouping, use any of the following Command-Line Parser E-mail aliases:

- parser-dev@cisco.com
- cli-output-police@cisco.com
- parser-police@cisco.com

27.8.1 Linking Commands

How do you actually do this linking? Let's use the example of a `foo_chain.c` file. Here's its basic skeleton:

```
#include "../parser/macros.h"
<and what ever other includes you need>

#include "../foo/parser_defs_foo.h"
This include file would define constants used by foo's CLI files and its
action routines.

/*
 * foo config commands
 */
#define ALTERNATE NONE
#include "cfg_foo.h"
LINK_POINT (foo_commands, ALTERNATE);
#undef ALTERNATE
```

`cfg_foo.h` has the parse nodes and your well-commented syntax. It can contain a set of commands, not just one command. To tell the parser to add this set of commands to its parse tree, you do this:

```
parser_add_commands (PARSE_ADD_CFG_LAST_CMD,
&pname(foo_commands), "foo cfg commands");
```

The `&pname(foo_commands)` essentially points to the base of your commands. The `PARSE_ADD_CFG_LAST_CMD` tells the parser where you want this set in the parse tree, that is, the order the command should appear when someone does a **show running**. In this case it would be added to the global commands that appear after all the interface configuration commands.

Adding Commands Dynamically

```

/*
 * foo show commands
 */
#define ALTERNATE NONE
#include "exec_show_foo.h"
LINK_POINT (foo_show_commands, ALTERNATE);
#undef ALTERNATE

```

Same thing here, for the **show** commands:

```

parser_add_commands (PARSE_ADD_SHOW_CMD,
                     &pname(foo_show_commands), "foo show commands");

```

You continue in this fashion for other types of commands. The point is that you group them according to where you want them in the parse tree and then you tell the parser where to put them.

Of course, `parser_add_commands()` is typically called from some initialization function; it's not strung out through the `foo_chain.c` file, as shown above. And there is a shorthanded way of adding a number of commands using a table. First define the table:

```

static parser_extension_request foo_init_table[] = {
{ PARSE_ADD_CFG_LAST_CMD, &pname(foo_commands) },
{ PARSE_ADD_SHOW_CMD, &pname(foo_show_commands) },
{ (parser_chain_id(LIST_END)), NULL },
};

```

and then add them, using just one call:

```

parser_add_command_list(foo_init_table, "foo");

```

If your commands are not required except under certain conditions, then you should only add them if that condition is true. Unfortunately, you can't take them out once they've been added, if that condition happens to turn false. Nevertheless, don't just blindly add them in a subsystem init function, unless they should be there from the get-go.

27.9 Adding 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.

27.9.1 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 )

```

27.9.1.1 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,
        "loption", "First option", PRIV_ROOT);
```

In 12.2 and earlier releases:

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
};
```

In 12.3 and later releases:

To uniquely identify the link point, you must create a global address label using the `chainiddef` macro from `config.h` in a .c source file:

```
chainiddef(LIST_END);
```

and declare the new label in a .h file, using:

```
chainiddecl(LIST_END);
```

Note Do not add an enum to `h/parser.h`.

27.9.2 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);
```

27.9.2.1 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));
```

27.9.3 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:
  Name          ID   Addr      Type
  end of list   0    0x0       1
  exec          1    0x5CF04   1
  .
  .
```

Adding Commands Dynamically

option command	XXX	0xXXXXXX	1
----------------	-----	----------	---

Note In Release 12.2S and also 12.2 and 12.3-based branches, you need to add `#define PARSER_DEBUG_LINKS` in `/vob/ios/sys/h/parser.h` (toward the bottom of the file) in order to make the **show parser links** command available.

27.9.4 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);
```

27.9.4.1 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)},
    {(parser_chain_id(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
```

27.9.5 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);
```

27.9.5.1 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)},
    {parser_chain_id(LIST_END), NULL}
};

void isdn_parser_init (void)
{
    parser_add_command_list(isdn_chain_init_table, "ISDN");
}
```

27.10 Useful Parser Commands

There are some parser commands that are especially useful for developers. For example, you can list the parser commands that have been created for platform you are working on by using the show parser commands, as shown in Table 27-7. See also “Debugging Parser Ambiguity” for information on the **parser debug ambig** command.

Table 27-7 Useful show parser Commands

Command	Description
show parser modes	Lists the names of all the modes on the platform.
show run	This command is hard coded to work only for privileged users in level 15 to get a complete running configuration

CCO has a command lookup page that is quite useful at times:

<http://www.cisco.com/cgi-bin/Support/Cmdlookup/ios-command-lookup.pl>

You need a cisco.com account to access this webpage.

27.11 Guidelines for Internal and Hidden Commands and More

Hidden commands are defined by the `PRIV_USER_HIDDEN` and `PRIV_HIDDEN` privilege options. The only continuous hidden command should be the **service internal** global configuration command discussed in section 27.11.1 “Use Internal Commands, Not Hidden.”

The second usage for hidden commands is to hide a deprecated command for a single release. This is to allow customers to migrate to a new release without breaking any scripts immediately. These deprecated hidden commands should be removed in the subsequent releases.

27.11.1 Use Internal Commands, Not Hidden

Other commands which are not to be generally used by the network operators should be defined as internal commands, that is, with the `PRIV_INTERNAL` privilege option. This set of commands is enabled via the **service internal** global configuration command. These commands must be documented at the same level as all other exposed commands.

Customers should not be “playing with unknown commands” on live networks. It is recommended that any `PRIV_INTERNAL` commands generate a banner saying something to the effect of “Beware: there be dragons here,” without requiring a confirmation from the operator.

27.11.2 One-Off Commands Treated as Normal

Commands that are supposedly only wanted by one customer, or are promised to have a very short lifespan, should still be visible and documented for the not-unlikely event that they live longer or are more widely used than was initially expected.

27.11.3 Why Hidden Commands are Discouraged

The reason hidden commands are discouraged is that you don’t get help with hidden commands, which is particularly onerous since the hidden commands are often the least used and least intuitive of all.

Adding new hidden commands (for example for debugging) is tempting to developers, but is usually not the best alternative because it means TAC can’t answer questions about those commands, and it is harder for other development and TAC engineers to learn about and use those commands. Presumably those commands are useful, or else they shouldn’t be in the image in the first place.

27.11.4 Guidelines for Confirmations

Commands that ask for confirmation are discouraged as they break automatic scripts. Only seriously destructive commands, such as the **reload** and the **copy run start**-enabled commands, should include confirmations.

27.11.5 Permanently Hidden Commands

To find all hidden (temporary and permanent) commands, **grep** for the HIDDEN flag in the parser's tree top node macro (all commands underneath the HIDDEN flag are hidden). Here are some of the hidden commands that are kept hidden permanently to protect internal product information:

- 1 **memory check-interval**
- 2 **memory malloc-list-use-malloc**
- 3 **memory try-malloc-lite**
- 4 **memory validate-checksum**
- 5 **show chunk**
- 6 **write core**

27.12 Useful Exit Functions and Submode Attributes

This section discusses how and why exit from a submode might occur, and describes some attributes related to command processing in submodes.

27.12.1 Submode Exit Functions

The following functions work to provide exit functionality for the CLI.

To exit all configuration submodes from the current submode depth of a session block, call the [`exit_all_config_submode\(\)`](#) function.

```
void exit_all_config_submode (parseinfo *csb);
```

To return to the top-level `config` mode from a mode that is one level beneath it, call the [`exit_config_submode\(\)`](#) function.

```
#include "config.h"
void exit_config_submode (parseinfo *csb);
```

27.12.2 Exiting a Submode

A submode is exited for many reasons including:

- Issuing the **exit** command in the submode.
- Issuing the **end** command in the submode.
- Entering a command that does not match the current submode, but matches a command in the “alternate” or parent mode of the submode.

This is an implicit feature of the parser that allows the flexibility to configure any parent configuration mode command from a child mode, although this has the side effect of changing the current mode to the parent mode as a result. See Section 27.12.3, “Implicit Submode Exit to Parent Mode”, for details and examples.

When the submode is exited for any of the reasons above, the parser invokes the `submode_exit_func()` pointer in the CSB. The submode implementation code has to provide the pointer to the `submode_exit_func()` pointer during the `save_var` function vector operation for the given submode (see the `parser_add_mode()` API reference page), and reset the `submode_exit_func()` during the `reset_var` function vector operation for the given submode.

For example:

```
void this_submode_exit_func (parseinfo *csb)
{
    /*
     * This function is invoked whenever this submode
     * is exited
     */
}

void *submode_save_var (parseinfo *csb)
{
    ...
    csb->submode_exit_func = this_submode_exit_func;
    ...
}

void submode_reset_var (parseinfo *csb, void *var)
{
    ...
    csb->submode_exit_func = NULL
}
```

Note The submode context (typically `csb->userDataBlock`) is not always available when the `submode_exit_func()` is invoked. Therefore, the context has to be saved as part of the `save_var` and restored as part of the `reset_var` function vectors.

27.12.3 Implicit Submode Exit to Parent Mode

The parser implements an implicit exit mechanism in which the parser automatically attempts to find and execute a command in the parent (or alternate) mode when the command failed in a submode. If the command is valid in the parent mode, the parser exits the submode and enters the parent mode. This happens internally and automatically, rather than as a result of explicitly entering an exit command or calling a parser API exit function.

For example, the **service internal** command is a configuration mode command that can still be executed in the `config-vlan` submode on a router. If this command is entered when in `config-vlan` submode, at the end of the command execution, the router's mode is automatically changed to configuration mode (note the change in the command line prompt to `(config)#`):

```
cp-BL13-2-a(config-vlan)#service internal
cp-BL13-2-a(config)#
```

In the following example, the command **ipv6 mld snooping** runs in the parent mode (`config`) because the command is not found in the current submode (`vlan`), and the `(config)#` prompt shows the mode has changed to the parent mode:

```
cp-BL13-2-a(config)#vlan 222
cp-BL13-2-a(config-vlan)#ip?
% Unrecognized command
cp-BL13-2-a(config-vlan)#ipv6 mld snooping
cp-BL13-2-a(config)#ip?
ip ipc ipv6 ipx
```

If a command is invalid in a submode *and* in that submode's parent mode, the parser does not change the mode, and the session stays in the current submode. In the following example, entering the **encap ip** command while in **config-vlan** mode is unsuccessful because this command is not supported in either **config-vlan** mode or the parent mode, **config**:

```
cp-BL13-2-a(config)#vlan 222
cp-BL13-2-a(config-vlan)#encap ip
^
% Invalid input detected at '^' marker.
cp-BL13-2-a(config-vlan)#

```

27.12.4 The exit Command from a Submode

The **exit** command from a parser submode brings the parser to the alternate or parent mode.

For example

```
Router#conf t
Enter configuration commands, one per line. End with CNTL/Z.
Router(config)#interface Loopback 0
Router(config-if)#exit
Router(config)#

```

The **exit** command brings the parser from **interface** mode to **config** mode.

Here **interface** mode is the submode and **config** mode is the alternate or parent mode for the **interface** mode.

This **exit** command needs to be parsed in a separate CLI file for each mode.

For example:

```
Router(config)#interface Loopback 0
Router(config-if)#exit

```

This **exit** processing is handled in a separate file called `cfg_int_exit.h`.

Consider the router submode:

```
Router(config)#router bgp 101
Router(config-router)#exit

```

This **exit** processing is handled in a separate CLI file called `cfg_router_exit.h`.

27.12.5 Interface Submodes and Interface Ranges

When entering a submode under an interface, the submode command implementation must check whether the command is being invoked in the context of interface range, and set `INTF_RANGE_COND_INTF_SUBMODE` by invoking the following function:

```
intf_range_set_condition(csb, INTF_RANGE_COND_SUBMODE);
```

This makes sure that the interface submode is properly handled in the context of interface range.

27.12.6 Limiting Parser Command Searches in Interface Submodes

A specific implementation of an interface submode may require that the parser is prevented from looking for a command in the parent mode. For example, the **vrrs** command in `subinterface config` mode appears as follows:

```
Router(config-subif)# vrrs follow <string>
```

And in the global config mode as follows:

```
Router(config)# vrrs <string>
```

If a user makes the following entry in the `subinterface config` mode:

```
Router(config-subif)# vrrs follow
```

The parser falls back to `global config` mode and executes **vrrs <string>**, taking the **follow** keyword of the `subinterface config` mode as a *string* in the `global config` mode.

To prevent the parser from searching the submodes, set the following flag in the `EVAL()` macro.

```
csb->flag |= CONFIG_NO_LEVEL_CHANGE
```

In the example described above, the `CONFIG_NO_LEVEL_CHANGE` flag should be set in an `EVAL()` macro *after* the match of the **vrrs** keyword.

Here is the code example for the `EVAL()` macro for the **vrrs** interface:

```
EVAL( vrrs_int_no_level_change, vrrs_int_nvgen,
      csb->flags |= CONFIG_NO_LEVEL_CHANGE);

KEYWORD( vrrs_interface, vrrs_int_no_level_change, NONE,
         "vrrs", "VRSS Interface configuration commands",
         PRIV_CONF|PRIV_SUBIF );
```

27.13 Manipulating Parser Modes

This section describes how to add parser modes and aliases to parser modes.

27.13.1 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.

Note The `reset_vars()` and `save_vars()` functions should be used to reset and save state changes (if any) in the `(parseinfo)csb` structure. A NULL pointer should be passed for the `reset_vars()` and `save_vars()` functions only if there are no changes to the `csb`. When a `csb` state change is not reset correctly and `csb` changes the command mode to execute a command, it could result in the crash of the route processor. This is because the command mode may try to access the `csb`, expecting it to be in a different state.

27.13.1.1 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));
```

27.13.2 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,
                      const char *command);
```

27.13.2.1 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");
```

27.14 CLI Safely Using Data Shared with Interrupt Level Code

If a CLI accesses a structure or list that is also used at interrupt level, perform the following steps:

- For structures, it is recommended to:

Step 1 Allocate memory for a “shadow” structure:

```
shadow = malloc(sizeof(original_structure))
```

Step 2 Raise interrupts:

```
prev_level = raise_interrupt_level(ALL_DISABLE)
```

Step 3 Copy the data from the real structure to the shadow:

```
bcopy(original_structure, shadow, sizeof(orginal_structure))
```

Step 4 Restore the previous interrupt level:

```
restore_interrupt_level(prev_level)
```

Step 5 Use data from the shadow structure:

```
printf(format, .....
```

- For IDB lists only, see section 6.6.4, “Iterate a List of Private IDBs Safely,” in the “Interfaces and Drivers” chapter.
- For all other lists and queues, see section 22.8, “Iterate a List or Queue Shared with Interrupt Level Code,” in the “Queues and Lists” chapter.

27.15 Debugging Parser Ambiguity

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. The **debug parser ambiguity** command lets you analyze how a command is parsed, providing output of the ambiguity of the parse tree, useful for checking suspected parser chain breakage and understanding how ambiguous commands are handled by the parser.

See also ENG-91860 for information about situations where the output of **parser debug ambig** is used.

Before issuing the **debug parser ambiguity** command, the **service internal** command should be given in the configuration mode. The **debug parser ambiguity** service internal command (in older versions, it is a hidden command) is enabled in EXEC mode to analyze how a CLI command is parsed. The debug parser flag is set and a message will be displayed as “Parser ambiguity debugging is on.”

27.15.1 Content of Debug Parser Ambiguity Output

If a command is given with the `debug parser ambig ON`, the keywords which are traversed by the parser are displayed:

- Step 1** The parser begins at [top] node. If the top node, for example, “no,” does not match the token input, for example, “configure,” then a message “Didn’t match keyword “no”” is displayed. This message is displayed for each keyword that does not match with the input token. Once the match occurs, a message “Matched keyword “configure”” is displayed.
- Step 2** The output shows, at which point of the parse tree traversal, each token is matched with a keyword and when, a no-alt, ALTERNATE, NONE or an EOL action is performed.
- Step 3** The content of the ambig-buffer and ambig-save buffer are displayed which is helpful to understand how ambiguity commands are checked. Two buffers ambig-buffer and ambig-save buffer are used by the parser to check for ambiguity. When a command is being parsed and if a first match occurs, the match is stored in the ambig-buffer and then copied to the ambig-save buffer. If a second match occurs, it is stored in the ambig-buffer and so the parser can check for ambiguity using the buffers, as the ambig-save buffer will contain the first match.
- Step 4** The value of `csb->visible_ambig_multiple_funcs` and `csb->hidden_ambig_multiple_funcs` are shown whenever a `no_alt` action takes place. From this value, it is possible to know how many keywords are matched for a given token. If the value of `csb->visible_ambig_multiple_funcs` is 0, then, for the given command, no match has occurred. If the value is 1, then exactly, one match has occurred. If the value is greater than 1, then it means more than 1 match has occurred, that is, an ambiguity has occurred.

- Step 5** When the command is traversed for help, then the current help string stored in the buffer is shown in the output.

27.15.1.1 Actions Performed When a no_alt node or EOL Is Encountered

Whenever a no_alt action is called, several actions are performed, depending on the state of the parser:

- The state of the parser is identified using ambig-buffer and ambig-save buffer.
- Based on the state of the parser, visible/hidden multiple_funcs is changed incrementally.

In the output of debug parser ambig, we come across statements such as:

```
Increasing visible multiple_funcs to 1 in no_alt_action 3
Increasing visible multiple_funcs to 2 in no_alt_action 2
```

When a no_alt node is encountered, depending upon the state of the parser, the visible multiple_funcs is incremented due to

- no_alt_action 1—Priv set processing, that is, csb->priv_set is not PRIV_NULL.
- no_alt_action 2—The contents of ambig-buffer and ambig-save buffer are not the same and the content of ambig-buffer is copied to ambig-save buffer.
- no_alt_action 3—csb->help is set and a help command is being parsed.

The value of csb->multiple_funcs is checked by parse_cmd() to determine if an error has occurred. Its value will be zero if no command match occurred. It will be one if exactly one command is matched or if it is an unambiguous help command. It will be greater than one if the command input is ambiguous.

In the output of debug parser ambig, we also come across statements such as

```
Ambig case 1 or
Ambig case 3 etc.
```

Depending upon the state of the parser, the no_alt action displays the ambig case number, from which inferences can be made. If the ambig case is 1, it means the contents of ambig-buffer and ambig-save buffer are different and the multiple_funcs is incremented by the no_alt action. Similarly, if the ambig case is 2, it means that the content of both the buffers are the same. If the ambig case is shown as 3, it means this is the first match and multiple_funcs has already been set. If the ambig case is 4, it means the ambig-buffer is empty and the end of the parse chain is attained.

When an EOL is encountered, the debug parser ambig displays a statement “EOL action.” The Eol-action function in turn calls the no_alt action. It also shows statements such as

```
Increasing visible multiple_funcs to 1 in eol_action 1 or
Increasing visible multiple_funcs to 1 in eol_action 2
```

The reason is stated as follows:

- eol_action 1—if csb->priv_set is TRUE, that is, the keyword is set to this Privilege (for privilege commands).
- eol_action 2—if eol is matched, that is, the user has hit RETURN and it is not a help command.

27.15.1.2 Example Analysis of the Output of debug parser ambig

A simple command `conf t`, is given at the EXEC mode to change the mode to config mode and the output of `debug parser ambig` is analysed as follows:

Output:

```

00:12:04: Didn't match keyword 'clear'
00:12:04: Didn't match keyword 'debug'
00:12:04: Didn't match keyword 'no'
00:12:04: Didn't match keyword 'exit'
00:12:04: Didn't match keyword 'quit'
00:12:04: Didn't match keyword 'logout'
00:12:04: Didn't match keyword 'lock'
00:12:04: Didn't match keyword 'profile'
00:12:04: Didn't match keyword 'unprofile'
00:12:04: Didn't match keyword 'set'
00:12:04: Didn't match keyword 'show'
00:12:04: Didn't match keyword 'terminal'
00:12:04: Didn't match keyword 'test'
00:12:04: Didn't match keyword 'undebbug'
00:12:04: Didn't match keyword 'disconnect'
00:12:04: Didn't match keyword 'name-connection'
00:12:04: Didn't match keyword 'resume'
00:12:04: Didn't match keyword 'where'
00:12:04: Matched keyword 'configure'.....Match with keyword
"configure"
00:12:04: csb->command_visible = 1
00:12:04: saving string 'configure' in ambig buffer'....Start traversing in
accept path and the matched string is stored in the ambig-buffer.
00:12:04: Matched keyword 'terminal'.....Match with "terminal"
00:12:04: csb->command_visible = 1
00:12:04: saving string 'terminal' in ambig buffer 'configure '
00:12:04: Eol Action.....Match with EOL
00:12:04: Increasing visible multiple_funcs to 1 in eol_action 2
00:12:04: saving string '(cr)' in ambig buffer 'configure terminal '
00:12:04: 1 Ambig buf 'configure terminal (cr)'.....no_alt called in EOL
action.
00:12:04: 1 Ambig_save buf ''
00:12:04: Ambig case 3.....Ambig case 3 shows
that this is the first match.
00:12:04: 2 Ambig buf ''
00:12:04: 2 Ambig_save buf 'configure terminal (cr)'
00:12:04: csb->visible_ambig.multiple_funcs = 1
00:12:04: csb->hidden_ambig.multiple_funcs = 0
00:12:04: Current help string:.....Since help is not
requested, help string is empty.
00:12:04: Didn't match keyword 'memory'.....|  

00:12:04: Didn't match keyword 'overwrite-network'.....| Searching the
alternates of | "config" to detect ambiguity.
00:12:04: Didn't match keyword 'network'.....|
00:12:04: 1 Ambig buf 'configure '
00:12:04: 1 Ambig_save buf 'configure terminal (cr)'....There is no alternate
after network, so a no_alt action is called
00:12:04: Ambig case 2.....Ambig case 2 shows
that the content of both buffers are same and hence no ambiguity.
00:12:04: 2 Ambig buf 'configure '
00:12:04: 2 Ambig_save buf 'configure terminal (cr)'|
```

```

00:12:04: csb->visible_ambig.multiple_funcs = 1.....multiple_funcs is not
incremented because, a second match has not occurred. so, there is no
ambiguity.
00:12:04: csb->hidden_ambig.multiple_funcs = 0
00:12:04: Current help string:
00:12:04:
00:12:04: Didn't match keyword 'disable'
00:12:04: Didn't match keyword 'enable'
00:12:04: Didn't match keyword 'ping'
00:12:04: Didn't match keyword 'systat'
00:12:04: Didn't match keyword 'traceroute'
00:12:04: Didn't match keyword 'who'
00:12:04: Didn't match keyword 'write'
00:12:04: Didn't match keyword 'hangup'
00:12:04: Didn't match keyword 'login'
00:12:04: Didn't match keyword 'reload'
00:12:04: Didn't match keyword 'send'
00:12:04: Didn't match keyword 'setup'
00:12:04: Didn't match keyword 'more'
00:12:04: Didn't match keyword 'format'
00:12:04: Didn't match keyword 'erase'
00:12:04: Didn't match keyword 'rmdir'
00:12:04: Didn't match keyword 'mkdir'
00:12:04: Didn't match keyword 'cd'
00:12:04: Didn't match keyword 'pwd'
00:12:04: Didn't match keyword 'dir'
00:12:04: Didn't match keyword 'undelete'
00:12:04: Didn't match keyword 'delete'
00:12:04: Didn't match keyword 'copy'
00:12:04: Didn't match keyword 'rename'
00:12:04: Didn't match keyword 'snmp'
00:12:04: Didn't match keyword 'ncia'
00:12:04: Didn't match keyword 'sdlc'
00:12:04: Didn't match keyword 'help'
00:12:04: 1 Ambig buf ''.....no_alt is called at
the end.
00:12:04: 1 Ambig_save buf 'configure terminal (cr)'
00:12:04: Ambig case 4.....The ambig-buffer is
empty and the end of parse chain is attained.
00:12:04: 2 Ambig buf ''
00:12:04: 2 Ambig_save buf ''
00:12:04: csb->visible_ambig.multiple_funcs = 1.....Exactly one match has
occurred
00:12:04: csb->hidden_ambig.multiple_funcs = 0
00:12:04: Current help string:
00:12:04:
gt4-7200-2(config)#.....Command is executed
and the mode has got changed from EXEC to Config mode.

```

The following steps outline the debug parser actions:

- Step 1** The parser begins with the top node, in this example, the top node is “clear.” As it did not match with the token “conf,” a message displays; “Didn’t match keyword clear.” The same message is displayed for the alternatives such as “debug,” “no,” “exit,” etc., until it matches with the keyword “configure.” A message “Matched Keyword configure” is displayed. The string “configure” is saved in the ambig_buffer.

- Step 2** The parser proceeds in the accept direction, that is, it matches the next input token “t” with the options of the **configure** command. The match occurs and another message displays “Matched keyword terminal.” After this, because an EOL is matched, a message “Eol Action” is displayed and an EOL action is called. As explained above, a statement “Increasing multiple_funcs to 1 in eol_action 2” displays.
- Step 3** The string “terminal” is concatenated with “configure” in the ambig-buffer. Now the content of ambig-buffer is “configure terminal.” Because the ambig case is shown as 3, it means this is the first match and `multiple_funcs` has been set as one match has occurred.
- Step 4** The parser searches the remaining alternate nodes in the parse tree to identify ambiguous commands.
- Step 5** At the end, a `no_alt` is called, which shows the ambig case as 4; all the alternates are checked and the parse chain has come to an end. The value of `visible multiple_funcs` is shown as 1, which means that exactly one match has occurred and there is no ambiguity in the input command and the command will be executed.

27.15.1.3 Identification of Parser Chain Breakage From the Output of debug parser ambig

Parser chain breakage occurs due to the improper usage of `no_alt` node. A `no_alt` node should only be used when the end of list of alternative tokens is attained.

27.15.1.3.1 Example Parser Chain Breakage

Consider the DDTs CSCdm64639. The problem reported was:

“The `sh isis database ?` command includes redundant information. It displays the output modifier twice. Also, the `sh ip sdr ?` command displays only one option. It omits the rest of the options.”

The output of `debug parser ambig` is shown below:

```
gt4-7200-1#sh isis database ?
00:01:49: Didn't match keyword 'clear'
00:01:49: Didn't match keyword 'debug'
00:01:49: Didn't match keyword 'no'
00:01:49: Didn't match keyword 'exit'
00:01:49: Didn't match keyword 'logout'
00:01:49: Didn't match keyword 'lock'
00:01:49: Didn't match keyword 'profile'
00:01:49: Didn't match keyword 'unprofile'
00:01:49: Didn't match keyword 'set'
00:01:49: Matched keyword 'show'Matched with keyword "show"
00:01:49: csb->command_visible = 1
00:01:49: saving string 'show' in ambig buffer ''
00:01:49: Didn't match keyword 'aliases'Start traversing in the accept path
00:01:49: Didn't match keyword 'configuration'
00:01:49: Didn't match keyword 'running-config'
00:01:49: Didn't match keyword 'startup-config'
00:01:49: Didn't match keyword 'history'
00:01:49: Didn't match keyword 'hosts'
00:01:49: Didn't match keyword 'interfaces'
```

```
00:01:49: Didn't match keyword 'classmap'.....The "classmap" macro  
may have caused the parser chain breakage because immediately after trying to  
match with "classmap" a no_alt is being called, even before "isis" is  
matched.  
00:01:49: 1 Ambig buf 'show '  
00:01:49: 1 Ambig_save buf ''  
00:01:49: Ambig case 3 [Even before "isis" is matched, a no_alt is called]  
00:01:49: Increasing visible multiple_funcs to 1 in no_alt_action 3  
00:01:49: 2 Ambig buf ''  
00:01:49: 2 Ambig_save buf 'show '  
00:01:49: csb->visible_ambig.multiple_funcs = 1  
00:01:49: csb->hidden_ambig.multiple_funcs = 0  
00:01:49: Current help string:  
00:01:49:  
00:01:49: Didn't match keyword 'tdm'  
00:01:49: Didn't match keyword 'traffic-shape'  
00:01:49: Didn't match keyword 'syscon'  
00:01:49: Didn't match keyword 'bootvar'  
00:01:49: Didn't match keyword 'pas'  
..  
..  
..  
00:01:49: Didn't match keyword 'decnet'  
00:01:49: Didn't match keyword 'clns'  
00:01:49: Matched keyword 'isis'Match with keyword "isis"  
00:01:49: csb->command_visible = 1  
00:01:49: saving string 'isis' in ambig buffer 'show '..Start traversing in  
the accept path  
00:01:49: Matched keyword 'database'.....Match with keyword  
"database"  
00:01:49: csb->command_visible = 1  
00:01:49: saving string 'database' in ambig buffer 'show isis '  
00:01:49: Didn't match keyword 'level-1'  
00:01:49: Didn't match keyword '11'  
00:01:49: Didn't match keyword 'level-2'  
00:01:49: Didn't match keyword '12'  
00:01:49: Didn't match keyword 'detail'  
00:01:49: Didn't match keyword 'verbose'  
00:01:49: Eol ActionMatch with EOL  
00:01:49: saving string '(cr)' in ambig buffer 'show isis database '  
00:01:49: 1 Ambig buf 'show isis database (cr)'  
00:01:49: 1 Ambig_save buf 'show '  
00:01:49: Ambig case 2  
00:01:49: 2 Ambig buf 'show isis database (cr)'  
  
00:01:49: 2 Ambig_save buf 'show isis database (cr)'  
00:01:49: csb->visible_ambig.multiple_funcs = 1  
00:01:49: csb->hidden_ambig.multiple_funcs = 0  
00:01:49: Current help string:  
  
00:01:49: detail Detailed link state database information  
00:01:49: 11 IS-IS Level-1 routing link state database  
00:01:49: 12 IS-IS Level-2 routing link state database  
00:01:49: level-1 IS-IS Level-1 routing link state database  
00:01:49: level-2 IS-IS Level-2 routing link state database  
00:01:49: verbose Verbose database information  
00:01:49: | Output modifiers  
00:01:49: (cr)  
00:01:49:
```

Debugging Parser Ambiguity

```

00:01:49:
00:01:49: Eol Action
00:01:49: saving string '(cr)' in ambig buffer 'show isis database '
00:01:49: 1 Ambig buf 'show isis database (cr)'
00:01:49: 1 Ambig_save buf 'show isis database (cr)'
00:01:49: Ambig case 2
00:01:49: 2 Ambig buf 'show isis database (cr)'
00:01:49: 2 Ambig_save buf 'show isis database (cr)'
00:01:49: csb->visible_ambig.multiple_funcs = 1

00:01:49: csb->hidden_ambig.multiple_funcs = 0
00:01:49: Current help string:

00:01:49: detail Detailed link state database information
00:01:49: 11 IS-IS Level-1 routing link state database
00:01:49: 12 IS-IS Level-2 routing link state database
00:01:49: level-1 IS-IS Level-1 routing link state database
00:01:49: level-2 IS-IS Level-2 routing link state database
00:01:49: verbose Verbose database information
00:01:49: | Output modifiers
00:01:49: (cr)
00:01:49:
00:01:49:
00:01:49: Didn't match keyword 'topology'
00:01:49: Didn't match keyword 'route'
00:01:49: Didn't match keyword 'hostname'
00:01:49: Didn't match keyword 'spf-log'
00:01:49: Didn't match keyword 'mpls'

.

00:01:49: Didn't match keyword 'ncia'
00:01:49: Didn't match keyword 'sdlc'
00:01:49: Didn't match keyword 'help'
00:01:49: 1 Ambig buf ''
00:01:49: 1 Ambig_save buf 'show isis database (cr)'
00:01:49: Ambig case 4
00:01:49: 2 Ambig buf ''
00:01:49: 2 Ambig_save buf ''
00:01:49: csb->visible_ambig.multiple_funcs = 1
00:01:49: csb->hidden_ambig.multiple_funcs = 0
00:01:49: Current help string:

00:01:49: detail Detailed link state database information
00:01:49: 11 IS-IS Level-1 routing link state database
00:01:49: 12 IS-IS Level-2 routing link state database
00:01:49: level-1 IS-IS Level-1 routing link state database
00:01:49: level-2 IS-IS Level-2 routing link state database
00:01:49: verbose Verbose database information
00:01:49: | Output modifiers
00:01:49: (cr)
00:01:49:
00:01:49:
gt4-7200-1#

```

The problem may be due to the `sh classmap`, that is, in the `classmap` macro, the alternate node may be specified as `no_alt` instead of `ALTERNATE`. After matching the keyword “`show`” and before matching the keyword “`isis`” in the above output, a `no_alt` action is called. Now we can suspect that

the classmap macro is the cause for the parser chain breakage. To verify, the header file for the “show classmap” is checked. If the alternate path is not specified as no_alt, then some other hidden command is causing the parser chain breakage. To find out which hidden command causes that breakage, we should analyze the output of debug parser ambig for the command sh isis database, which is not a help command, and not show isis database ?, which is a help command. In this case, a no_alt action is called before “isis” is matched and the visible multiple_funcs is incremented to 1, even before matching “isis.” Also, before the EOL action is called for the command sh isis database ? TEST_MULTIPLE_FUNCS macro is used to check for the value of multiple_funcs. As a result, the chain breaks.

Consider the file /sys/clns/exec_show_isis.h:

```
EOLS      (show_isis_db_eol, show_isis_database, SHOW_ISIS_DATABASE);

STRING (show_isis_db_get_lspid, show_isis_db_opts, NONE, OBJ(string,1),
"LSPID in the form of xxxx.xxxx.xxxx.xx-xx or name.xx-xx");
IFELSE (show_isis_db_lspid, show_isis_db_get_lspid, NONE,
!*GETOBJ(string,1));

TEST_MULTIPLE_FUNCS(show_isis_db_test, show_isis_db_lspid,
show_isis_db_eol, NONE);

.....
.....
KEYWORD_ID(show_isis_db, show_isis_db_opts, show_isis_topo_kwd,
OBJ(int,1), ISIS_CIRCUIT_L1L2, "database",
"IS-IS link state database", PRIV_USER);

NOP (show_isis_kw, show_isis_db, NONE);

KEYWORD (show_isis_kw, show_isis_db, ALTERNATE,
"isis", "IS-IS routing information", PRIV_USER|PRIV_USER_HIDDEN);
```

The code that caused the parser chain breakage is as follows:

In the file /sys/qos/exec_show_classmap.h, the existing code is:

```
KEYWORD (show_classmap_kw, show_classmapname, no_alt,
"class-map", "Show QoS Class Map", PRIV_USER);
```

which breaks the parser chain due to the no_alt node. The problem is fixed by changing the code as follows:

```
KEYWORD (show_classmap_kw, show_classmapname, ALTERNATE,
"class-map", "Show QoS Class Map", PRIV_USER);
```

In the debug parser ambig output, when a no_alt action is called, even before a token is matched with a keyword, then it can be identified that the parser chain break has occurred. The keyword that was compared with the token, just before the no_alt action was called, should be suspected to have caused the parser chain break. The header file of that command should be checked and rectified. In the above example, the ?sh classmap? caused the chain break.

Proper usage of no_alt, ALTERNATE and NONE will not cause a parser chain break. For more illustrations, refer to bugs with ids CSCdm57165, CSCds21795,CSCds44678, CSCds44680.

Adding New Interface and Controller Types

27.15.1.4 When to Use the Output of debug parser ambig

The following list includes the most common uses of the debug parser ambig command output.

- 1 When a parser chain breakage is suspected, that is, when a command is executed, and the parser gives an unrecognized/incomplete command, use debug parser ambig to trace the point at which the chain break occurs.
- 2 **debug parser ambig** ouput is used to understand how the ambiguous commands are handled by the parser.
- 3 When help commands do not display the correct help strings, this output can be used to check the problem.
- 4 When new commands or new option to a command are added, debug parser ambig output can be used to understand the flow of the parser chain.

27.16 Adding New Interface and Controller Types

The Cisco IOS parser infrastructure was changed in 12.3T to support the need to add new interface types and remove the limitation of 64 maximum different types of interfaces and controllers per router. The new infrastructure provides support for string representation of the valid interfaces for the command and support for converting the string into a bit mask of size 192 at run time using a Boolean logic parser.

This section describes how to create new interface and controller types using the new infrastructure. See EDCS-249814 for more information on the design of the new Cisco IOS parser infrastructure for creating new interface types.

This section includes the following subsections:

- List of Macros and Functions
- Using `DEFINE_IFTYPE` to Add New Interface Types
- Using `DEFINE_CFTYPE` to Add New Controller Types

27.16.1 List of Macros and Functions

The following new macros and API functions were added to support the new infrastructure (see the *Cisco IOS API Reference* for details on these macros and functions):

- 1 `DEFINE_CFTYPE` (for Cisco IOS controller types like T1 and E1) to create the `iftype` entries for each controller's data structure.
- 2 `DEFINE_IFTYPE` (for Cisco IOS interface types like Ethernet and Fast Ethernet) to create the `iftypes` entries for each interface's data structure.
- 3 `parser_add_cftype_symbol()` developed to add the structure (created by the `DEFINE_CFTYPE` macro) to the respective linked list that now replaces the `cftypes[]` array.
- 4 `parser_add_iftype_symbol()` developed to add the structure (created by the `DEFINE_IFTYPE` macro) to the respective linked list that now replaces the `iftypes[]` array
- 5 `parser_add_iftype_group()` developed to add a group of structures (created by the `DEFINE_IFTYPE` macro) to the respective linked list

27.16.2 Using DEFINE_IFTYPE to Add New Interface Types

Complete the following steps to add new interface types for XYZ features:

- Step 1** Add the `DEFINE_IFTYPE` macro in your subsystem-specific subsys init file (`xyz_init.c`) for the new interface. For example:

```
DEFINE_IFTYPE(XYZ, xyz-interface, interface, 0,
              MAXINT, 0, 0, NO_VC, NO_VC, TRUE, FALSE, BEFORE_HW_INTS, 0, 0,
              0xFFFFFFF, TRUE, FALSE)
```

- Step 2** Use the `parser_add_iftype_symbol()` function to register the new interface with the parser, as shown in `xyz_init.c` here:

```
xyz_subsys_init ( )
{
    parser_add_iftype_symbol(PTYPE(XYZ));
}
```

Note If you do not have a relevant sub-system for the new interface type, add the macro and the API function call to the `parser/parser_iftypes.c` file.

- Step 3** Add the `IIFTYPE_XYZ` interface type in an `INTERFACE` macro defined in the `cfg_int_xyz.h` file, as shown here:

```
INTERFACE(name, accept, alternate, OBJ(idb,1), IIFTYPE_XYZ);
```

The `IIFTYPE_XYZ` string that is given as input to the parser is converted to the bit mask representation and the mask is stored in the CSB (Console Status Block) during parsing.

When the `PARSE_UNIT_ONLY` flag is specified in the `INTERFACE` macro, only a single type interface is allowed for parsing and if multiple types of interfaces are given as input an error message is displayed.

- Step 4** Include the `cfg_int_xyz.h` file in the `xyz.c` chain file.

- Step 5** Declare the `IIFTYPE_XYZ` interface type in the `xyz.c` chain file using the `DECLARE_IIFTYPE` macro, as shown here:

```
DECLARE_IIFTYPE(XYZ)
```

- Step 6** If you are using the `IIFTYPE_XYZ` interface name in any code, declare the general interface name using the `DECLARE_IFNAME` macro that will create the extern declarations for the `IFNAME_XYZ` general interface name:

```
DECLARE_IFNAME(XYZ)
```

Note Do not include `parser/parser_defs_parser.h` to get the extern declarations.

27.16.3 Using DEFINE_CFTYPE to Add New Controller Types

Complete the following steps to add a new XYZ controller type:

- Step 1** Add the `DEFINE_CFTYPE` macro in your subsystem-specific subsys init file (`xyz_init.c`) for the new controller. For example:

```
DEFINE_CFTYPE(XYZ, x cont, XYZ controller)
```

- Step 2** Use the `parser_add_cftype_symbol()` function to register the new interface with the parser, as shown in `xyz_init.c` here:

```
xyz_subsys_init ( )
{
    parser_add_cftype_symbol(CFTYPE(XYZ));
}
```

Note If you do not have a relevant sub-system for the new controller type, add the macro and the API function call to the `if/if_controller_iftypes.c` file.

- Step 3** Include the `cfg_xyz.h` file in the `xyz.c` chain file.

- Step 4** Declare the `IFTYPE_XYZ` interface type in the `xyz.c` chain file using the `DECLARE_IFTYPE` macro, as shown here:

```
DECLARE_IFTYPE(XYZ)
```

27.17 ICD (Intelligent Config Diff)

ICD is a feature that was added to Cisco IOS in Release 12.3T. See EDCS-228750 for functional specification information. The ICD feature is used like the UNIX diff command to compare two IOS config files in order to determine the changes that have been completed between them, and ICD is used most notably to support the IOS Rollback feature.

The ICD feature provides support in IOS for the following:

- 1 Network administrators can remotely determine what has changed in the running-config by using the following ICD exec command:

```
show archive differences file1 file2
```

- 2 Network administrators can remotely initiate an IOS Config Rollback—this in turn will trigger the ICD Algorithm:

```
config replace newconfig
```

- 3 Provisioning applications can use the ICD programmatic API to determine if the IOS running-config has changed. The application can then choose to perform a rollback or apply an incremental diff of a new IOS configlet.

An ICD Diff:

- 1 Produces an accurate list of differences, including context mode/sub-mode names, between two config files.
- 2 Produces a complete list of differences between the two files.
- 3 Represents all lines as additions or deletions.

- 4 Compares config files on any readable file media supported by IOS.
- 5 Aborts if an image is supplied instead of a config file.
- 6 Lists all the lines missing from each config file.
- 7 Returns error codes, and uses an API so internal applications can invoke the diff engine.
- 8 Supports IOS configs of different sizes, and with different contents. The ICD comparison engine does not make any distinction between configs and configlets.
- 9 Provides the ability to post-process its output:
 - (a) One form of post-processing is “Command suppression” - users may not wish to see some commands in the ICD output, e.g. “Version 12.2” or “hostname blah”. Users/applications should be allowed to submit a file containing a list of commands that should be filtered out from the final ICD output.
 - (b) Another form of ICD post-processing is command replacement. If replacement is needed, the application or user must provide the replacement algorithm. An example application that will use the replacement option is “IOS Config Replace”, which will provide an algorithm to “undo” changes found in the running-config.

Note ICD will read each file in its entirety into system RAM (Random Access Memory). RAM supports arbitrary seeks within a file. Hence copying the file into RAM from other media types means that ICD needs to only worry about file operations on RAM, simplifying the permutations of cases it needs to handle.

27.17.1 The ICD CLI

The commands to create ICD Diff output are entered in exec mode.

The ICD Diff command is shown here:

```
Router#show archive differences file1 file2
```

file1 is the startup-config or a previously saved config.

file2 is the running-config.

For example:

```
Router#show archive differences file1 file2
???
+
-
... snip ...
```

The ICD Incremental Diff command that lists only those lines from *file1* that are not present in *file2*, is shown here:

```
Router#show archive incremental_diffs file1 file2
```

file1 is the startup-config or a previously saved config.

file2 is the running-config.

For example:

```
Router#show archive incremental_diffs file1 file2
+???
```

```
+  
... snip ...
```

27.17.2 Providing ICD Support to Applications

The ICD feature is used in applications, such as IOS Config Rollback, for network management. To allow applications to use the ICD programmatic API to determine if the IOS running-config has changed, complete the following:

Step 1 Include the ICD header file:

```
#include "../ui/cfg_diff.h"
```

Step 2 Add the `cfgdiff_retcod do_cfgdiff()` API function, which must be called in order to generate the ICD output:

```
extern cfgdiff_retcod do_cfgdiff(dynamic_buffer *file1,  
                                 dynamic_buffer *file2,  
                                 dynamic_buffer *suppress_cmds,  
                                 dynamic_buffer *output_cmds,  
                                 diff_operation op,  
                                 boolean verbose);
```

The `suppress_cmds` parameter is a list of commands that the caller wants removed from the formatted output. An example where this may be used is the “Version” command in IOS, which is a documentation command that tells the reader which version of IOS generated a config. Users may not be interested in this command, so they can suppress any output related to this command by adding the text “Version 12.2” or “version 12.0” into the `suppress_cmds` buffer.

The `output_cmds` parameter contains the list of changed commands between the two IOS config files.

The `op` parameter indicates the format of the Diff output, `diffs`, `rollback`, or `incremental_diffs`.

`diffs` means that the output will list each line added to `file1` with a preceding “+”, and each line missing from `file2` with a preceding “-”.

`incremental_diffs` means that the output will list only the lines added to `file1`. It does not precede these lines with any special character, but instead prints the lines as they appear in the `file1` config (inclusive of any submode names that contain that line).

`rollback` means that the output will list the negation of each changed command. If a command was added, this format prints the equivalent undo version. If a command was removed, this format lists the removed command in the output.

The `verbose` parameter indicates whether any output should be printed to `stdio` or not. If set to `TRUE`, informational output from the ICD Algorithm is piped to `stdio`. It is recommended that applications set this to `FALSE` unless the output is being captured for display to a user.

For example:

```
extern cfgdiff_retcod do_cfgdiff(file1, file2, Version,  
                                 incremental_diffs, FALSE);
```

- Step 3** If `verbose` is set to TRUE, the caller needs to provide the initialized `dynamic_buffer` structure (which can be initialized by calling `init_dynamic_buffer()`). The memory space for the buffer contents will be malloced by the `do_cfgdiff()` function when necessary, so the caller *must* also free the allocated memory by calling `free_dynamic_buffer()` when there is no further need for the buffer contents.

27.18 IOS Config Rollback

IOS Config Rollback is a feature that was added to Cisco IOS in Release 12.3T. See EDCS-237327 for functional specification information. The IOS Config Rollback feature is used for rolling back or undoing changes to the running-config to allow the running-config to revert to a previously saved state. The IOS Config Rollback feature is built on top of the ICD feature.

The IOS Config Rollback feature provides support in IOS for the following:

- 1 Network administrators can remotely undo changes in the running-config by using the following exec command to replace the running-config with the previously saved configuration:

config replace savedconfig

- 2 Network administrators can remotely initiate an IOS Config Rollback to replace the running-config with a new configuration:

config replace newconfig

The IOS Config Rollback feature:

- 1 Works for the majority of nvgenmed IOS configuration commands.

- 2 Can undo:

- (a) Commands that are nvgenmed with a “no” prefix
 - (b) Commands that are nvgenmed without a “no” prefix
 - (c) Order-dependent commands (e.g. “access-lists”, “boot” commands, etc.)
 - (d) Submodes that can be removed from the running-config

- 3 Will undo the contents of a physical interface submode, but will not remove the submode itself from the running-config (removing a physical interface submode requires manual removal of the hardware (linecard) on which the interface is mounted).

- 4 Cannot undo commands that are nvgenmed by default because they cannot be “undone” by adding a “no” or “default” prefix. For example, the **no ip mroute-cache** or **no ip route-cache** commands. These commands may show up automatically when a router is upgraded with a new image, even though the user made no explicit configuration change. The IOS Config Rollback feature is “taught” about these commands, so that it can handle them appropriately when trying to undo commands.

- 5 Results in minimal change to code in other IOS subsystems. The only permissible changes to push back to other subsystems are:

- (a) Commands that cannot be programmatically undone using the IOS Config Rollback algorithm.
 - (b) Commands that show up in the running-config and cannot be deleted without manual intervention.

- (c) Commands that prompt the user for additional changes. This is because of the complications that occur with a programmatic rollback of a command when the user is prompted for additional input during the rollback process.
- 6** Ignores certain commands that are of limited interest to the end user. For example:
- (a) Commands like the “Version” and “timestamp” command that change often, but are of limited interest to users when comparing configuration files.
 - (b) Some commands present in saved config files may not be recognized by the version of IOS running on the target router. Hence such commands, when encountered, result in a syntax failure. The Rollback algorithm in this design is a multi-pass algorithm where commands with syntax failures are discovered in the first couple of passes, and compiled into a “library” of failed commands. Such commands are suppressed from the rollback output of later passes to ensure that the rollback algorithm only attempts to apply syntactically acceptable commands.

Note When syntax-checking is widely supported in IOS, the Rollback algorithm will be able to syntax check before applying the first rollback pass, ensuring better performance, and fewer passes.

27.18.1 The IOS Config Rollback CLI

The commands to do an IOS Config Rollback are entered in exec mode.

The IOS Config Rollback command is shown here:

```
Router#config replace file list force
```

file is the file with the startup-config, a previously saved config, or a new config.

list is optional. If specified, *list* provides a list of all commands applied in each pass.

force is optional. If specified, *force* tells the IOS Config Rollback to *not* prompt the user. Otherwise, the user will be sent prompts:

- Before processing the config, with the following warning message:
“This will apply all necessary additions and deletions to replace the current running configuration with the contents of the specified configuration file, which is assumed to be a complete configuration, not a partial configuration. Enter Y if you are sure you want to proceed.”
- If the new file is either zero length or less than 50% of the size of the running-config.

For example:

```
Router#config replace nvram:startup-config
Total number of passes:1
Rollback Done
```

27.18.2 Providing IOS Config Rollback Support to Applications

IOS Config Rollback is planned to be used in applications, such as Policy Manager, and is especially useful for remote network management. To allow applications to use the IOS Config Rollback programmatic API to change the running-config, complete the following:

- Step 1** Provide ICD support for your application. See subsection 27.17.2, “Providing ICD Support to Applications”.

- Step 2** Add the `cfgdiff_retcode apply_rollback()` API function, which must be called in order to generate the rollback:

```
extern cfgdiff_retcode apply_rollback(char *target, boolean verbose);
```

The `target` parameter passed to the API is an IFS file handle pointing to the config file that you want to replace the running-config with.

The `verbose` parameter indicates whether any output should be printed to `stdio` or not. If set to `TRUE`, informational output about the IOS Config Rollback operation is piped to `stdio`. It is recommended that applications set this to `FALSE` unless the output about the progress of the IOS Config Rollback needs to be captured for display.

For example:

```
extern cfgdiff_retcode apply_rollback(nvram:startup-config, FALSE);
```

27.19 How to Find a Command's EOL

To find a command's EOL node without hunting for wumpus, follow the instructions in this section or at:

<http://zed.cisco.com/confluence/display/PARSE/Home>

Note “parser cache” needs to be disabled before using this method for identifying a command’s EOL node.

27.19.1 Overview

Given a particular command, you may choose to read the source code parse chains (with help by `gid32` or `cscope`) to find the supporting EOL node. Some commands can be hard to follow because they involve link points, which presents a challenge when searching for the supporting EOL node. Additionally, reading the source code when things are not behaving as expected is sometimes not as helpful as viewing the actual result from the parser. This section identifies a command’s EOL node using `gdb` with an image that you have already built, simply by rebuilding one `.o` file and relinking the image (this is fast and easy).

27.19.2 Rebuild IOS Image With One Minor Change

(Skip to the next subsection if the image is already built with `GDB_FLAG=-g`.)

Complete the following steps to rebuild your image:

- Step 1** Add a line containing “`../parser`” to `/vob/ios/sys/gdb_required`.

- Step 2** Remove the `parser.o` file, which has no debugging info. Enter on the command line:

```
cd obj-processor  
rm parser.o
```

- Step 3** Rebuild the IOS image. This rebuilds `parser.o` with debugging symbols, and relinks it to the existing objects to create the desired image - this is relatively quick and painless for existing images.

27.19.3 Find a Command's EOL node

Complete these steps to find a command's EOL node once you have the image that was already built with GDB_FLAG=-g or was built per the steps in subsection 27.19.2:

Step 1 Load the image on a router or start the executable for IOU, then connect to it with gdb.

Step 2 Load the symbols:

```
(gdb) sym image.sun
```

Step 3 On IOU, submit the following to gdb:

```
(gdb) handle SIGALRM nostop
(gdb) handle SIGALRM noprint
(gdb) handle SIGALRM nopass
```

Step 4 Turn off the parser cache, as shown here:

```
router#conf t
router(config)#no parser cache
router#end
```

Step 5 Break into gdb by entering the **break parse_cmd_init** command and then, entering the **continue** command:

```
(cisco-6.1.0-gdb) break parse_cmd_init
Breakpoint 1 at 0x606f64: file ../../parser/parser.c, line 1614.
(cisco-6.1.0-gdb) c
Continuing.
```

Step 6 Navigate to the desired IOS (sub)mode, by entering the **continue** command for any triggered breakpoints.

Step 7 Enter the target command and then the **continue** command at the breakpoint. For example:

```
router#show version

Breakpoint 1, parse_cmd_init (csb=0x2658a70) at ../../parser/parser.c:1614
(cisco-6.1.0-gdb) c
Continuing.
Cisco IOS Software, Solaris Software (UNIX-IS-M), Experimental Version
12.4(20051230:102020) [rpratt-iou_demo_nogdb 107]
Copyright (c) 1986-2006 by Cisco Systems, Inc.
Compiled Wed 11-Jan-06 17:00 by rpratt

ROM: Bootstrap program is Solaris

ipsla-source uptime is 0 minutes
System returned to ROM by reload
Running default software

Solaris Unix (Sparc) processor with 30501K bytes of memory.
Processor board ID 10521077
8 Ethernet interfaces
8 Serial interfaces
16K bytes of NVRAM.

Configuration register is 0x0
```

```
router#
```

- Step 8** Submit one empty command (for example, hit the enter key), but do *not* enter the continue command at the breakpoint. Instead, submit the following to gdb:

```
p csb->action_func
p csb->matching_node
info symbol <address from p csb->matching_node>
```

For example:

```
Breakpoint 1, parse_cmd_init (csb=0x2658a70) at ../parser/parser.c:1614
(cisco-6.1.0-gdb) p csb->action_func
$1 = (void (*)()) 0x628978 <show_version>
(cisco-6.1.0-gdb) p csb->matching_node
$2 = (struct transition_ *) 0x21b6b34
(cisco-6.1.0-gdb) info symbol 0x21b6b34
PARSER_show_version_eol in section .data
(cisco-6.1.0-gdb)
```

This example shows that the EOL node's command action function is "show_version" and that the EOL node name is *node_name* (show_version_eol in this example), which comes from the info symbol result (that is, PARSER_node_name).

- Step 9** Use gid32 or cscope to search for <*node_name*> or the command's action function. gid32 and cscope are both ready to use in the nightly snapshots at /auto/ios-snaps/*. The EOLNS node is easily found, as shown here:

```
cd /vob/ios/sys/obj-sp-unix/
gid32 show_version_eol
/vob/ios/sys/ui/exec_show_version.h:48:EOLNS(show_version_eol, show_version);
/vob/ios/sys/ui/exec_show_version.h:49:KEYWORD (show_version,
show_version_eol, ALTERNATE,
```

gid32 finished at Thu Jan 12 12:56:30

And the command's support function is also easily found. For example:

```
rpratt@nmtg-view1[134] gid32 show_version | grep "show_version ("
/vob/ios/sys/ui/exec.c:2346:void show_version (parseinfo *csb)
rpratt@nmtg-view1[135]
```

For function declarations that do not follow the Cisco convention of a space between the function declaration's name and open parenthesis, the function can still be found relatively easy by scanning the results of gid32 within a searchable buffer (such as emacs). You can use cscope to search for the function declaration as well. Here is an example using gid32:

```
cd /vob/ios/sys/obj-sp-unix/
gid32 show_version
/vob/ios/sys/as/vfcfs_file_show.h:47:KEYWORD (show_version,
show_DSPWARE_keyword, no_alt,
/vob/ios/sys/as/vfcfs_file_show.h:60:KEYWORD (show_MODULE_capabilities,
show_MODULE_capabilities_eol, show_version,
/vob/ios/sys/ui/exec_show_hardware.h:46:EOLNS (show_HARDWARE_generic_eol,
show_version);
/vob/ios/sys/ui/exec_show_version.h:48:EOLNS (show_version_eol,
show_version);
/vob/ios/sys/ui/exec_show_version.h:49:KEYWORD (show_version,
show_version_eol, ALTERNATE,
/vob/ios/sys/ui/exec_show_version.h:54:#define ALTERNATE show_version
```

```

/vob/ios/sys/toaster/rpmxf-rp/exec_show_pxf.h:642:EOLNS
(rpmxf_show_hardware_eol, show_version);
/vob/ios/sys/ui/exec.c:2342: * show_version
/vob/ios/sys/ui/exec.c:2346:void show_version (parseinfo *csb)

gid32 finished at Thu Jan 12 13:06:16

```

If you have questions or comments, send them to parser-questions@cisco.com.

27.20 NVGEN Enhancements

The **show running-config** command was sped up as part of the NVGEN enhancement project (for more information, see EDCS-271956). The changes were committed to 12.3TPI3 and 12.2S RLS5. To enable this feature, the **parser config cache interface** command needs to be issued. An attempt to mark the parse chains that were configured and later use this information to speed up NVGEN was done. It was not feasible at that point since it involved extensive co-operation from other components for the identification of side effects and for the identification of commands that show up in configuration by default (that is, without configuring them).

27.20.1 Config Checkpointing

Config checkpointing is functionality that is available in the major branches of *Whitneyx*. There are two types of config checkpointing available:

- 1 NVGEN config checkpointing occurs during the following commands:
 - (a) **write memory**
 - (b) **show running config**
 - (c) **write checkpoint**
- 2 Line-By-Line (LBL) config checkpointing contains user commands that were entered after the last NVGEN checkpointing. During NVGEN checkpointing, the contents in LBL are flushed. As a result, LBL acts as a temporary best-effort method until the next NVGEN checkpointing occurs. The commands in LBL are replayed *after* NVGEN config checkpointing. This is particularly useful if the router crashes before any NVGEN command.

27.20.1.1 NVGEN Config Checkpointing

The following bullet list describes the function and use of NVGEN config checkpointing:

- Checkpointing defined as **config-mode-exit-triggered** is OFF by default (that is, default is **no service config-checkpoint**).
- The **write checkpoint** command is available to manually trigger checkpointing before a **process restart**.
- Commands that generate the running-config (via NVGEN) get a free **write checkpoint** as a side effect command.
- When a user issues a **process restart** command, they will receive a warning that checkpointing is out of date and that they should proceed as follows:
 - Accept the default reply (NO)
 - Issue a **write checkpoint**, and then

- Perform the process restart

27.21 Warning Against an Interactive CLI

An interactive CLI is actively discouraged for all config commands—many boxes are controlled via scripts now, not humans—going interactive only slows things down (and does not work in non-interactive access modes).

If you think a *config* command is very dangerous, you should make it a two knob configuration process (sort of like how “service internal” protects the user from certain commands) instead of going the interactive route.

CLI interactive (yes_or_no, ...) configuration commands are not only script unfriendly, but now this interaction can sit holding the configuration lock until the human user pulls the trigger (or the script manages to stumble past it).

This big lock is a poor substitute for proper data structure protection by components, but it’s a reality so avoid/discourage/do-not-approve interaction in config mode commands.

CISCO HIGHLY CONFIDENTIAL

Warning Against an Interactive CLI

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.cisco.com/ios/itc/embmgmt/snmp/index.shtml>

Note If you have questions about designing, writing, and testing MIBs that are not answered by this chapter or the SNMP Web site, contact the interest-mib@cisco.com and [mib-consulting](mailto:mib-consulting@cisco.com) email aliases.

28.1 SNMP Overview

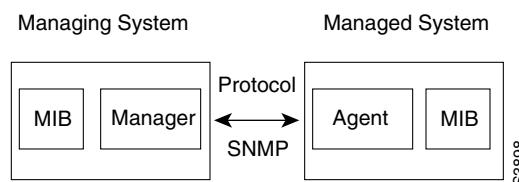
The section presents a high-level overview of SNMP. The discussion in this section is applicable to both SNMPv1 and SNMPv2. This section discusses the following topics:

- Internet Network Management Framework: Definition
- MIB: Definition
- SMI: Definition
- Transport Protocols
- SNMP Facilities
- Asynchronous Notifications

28.1.1 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). Figure 28-1 illustrates the Internet Network Management Framework model.

Figure 28-1 Internet Network Management Framework Model



28.1.2 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 know 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.

28.1.3 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.

28.1.4 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.

28.1.5 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.

28.1.6 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.

28.1.7 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

28.2 MIB Concepts

28.2.1 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 for SNMP.

28.2.2 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.

28.2.3 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.

28.2.4 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.

28.2.5 MIB Objects

28.2.5.1 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.

28.2.5.2 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.

28.2.5.3 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 identifier*. The final part of the OID, the *instance identifier*, 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.

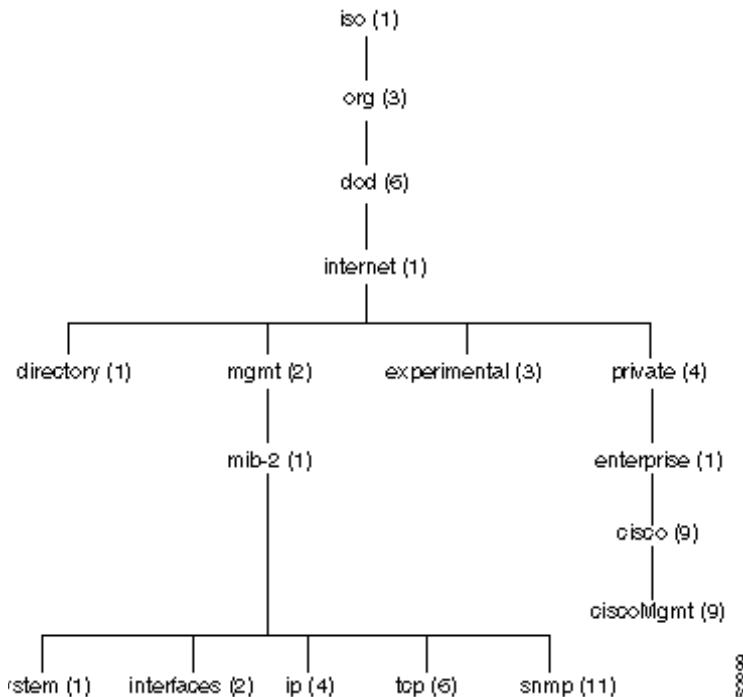
An object identifier is expressed as a series of integers or text strings. Technically, the numeric form is the *object name* and the text form is the *object descriptor*. In practice, both are called *object identifiers*, or *OIDs*. The numeric form is used in the protocols among machines. The text form, sometimes mixed with the numeric form, is for use by people.

Figure 28-2 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 sysDescr. Its full OID can be one of the following:

```
iso.org.dod.internet.mgmt.mib-2.system.sysDescr.0  
1.3.6.1.2.1.1.0
```

Figure 28-2 MIB Object Identifiers

28.2.6 SNMP Conceptual Tables

28.2.6.1 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.

28.2.6.2 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.

28.2.6.3 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 }
```

28.2.6.4 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.

28.2.6.5 Tables Inside of 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:

```
INDEX { 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.

```
INDEX { 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.

```
INDEX { rptrGroupIndex, rptrPortIndex }
```

28.3 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.

28.3.1 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.

28.3.2 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`.
- `RowIndexer`—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 `readOnly`.
- `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.

28.4 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. To contact the MIB police, use the `mib-police` email alias. The SNMP Infra Tool available at <http://mibportal:8080/SNMPInfraTool/index.html> includes a search capability that allows the user to search IETF, Cisco, and IEEE MIBs based on user entered keywords. Please refer to the *SNMP Infra Tool User Guide* for more information.
- 2 **Design**—The actual design of a MIB is discussed in the section “Design a MIB” later in this chapter.
- 3 **Review**—Any new MIB defined or updates to existing MIBs need to be reviewed and approved by the Cisco MIB police. For details on the MIB review process, see the section “MIB Review” later in this chapter.
- 4 **Implementation**—Implementing a MIB is discussed in section “Establish a New MIB” later in this chapter.
- 5 **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://wwwwin-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.
- 6 **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.
- 7 **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.

28.5 Design a MIB

28.5.1 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.

28.5.2 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.

28.5.3 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 IOS Embedded Management Group, which offers in-house SNMP MIB consulting services.

28.5.3.1 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.

28.5.3.2 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.

28.5.3.3 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.

28.5.4 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.

The SNMP Infra Tool at <http://mibportal:8080/SNMPInfraTool/index.html> includes a search capability that allows the user to search IETF, Cisco, and IEEE MIBs based on user entered keywords. Please refer to the *SNMP Infra Tool User Guide* for more information.

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.

28.5.5 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.

28.5.6 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.

28.5.6.1 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.cisco.com/ios/itc/embmgmt/mibpolice/process/cana_index.shtml

When a new MIB gets approved by the Cisco MIB police, a CANA number is automatically assigned to it.

28.5.6.2 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 1902—*Structure of Management Information for Version 2 of the Simple Network management Protocol (SNMPv2)* Obsolete, see RFC 2578.
- RFC 2578—*Structure of Management Information Version 2 (SMIV2)* RFC 2578 obsoletes RFC 1902.
- RFC 1903—*Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)* Obsolete, see RFC 2579.
- RFC 2579—*Textual Conventions for SMIV2* RFC 2579 obsoletes RFC 1903.
- RFC 1904—*Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)* Obsolete, see RFC 2580.
- RFC 2580—*Conformance Statements for SMIV2* RFC 2580 obsoletes RFC 1904.
- RFC 4181 - Guidelines for Authors and Reviewers of MIB Documents

The list of Cisco conventions and guidelines for MIB definitions can be found in EDCS 448894.

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.my`.
- **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*, *Index*, and *FramesSent*.
- *eventName*—A MIB event name, such as *ConnectionClosed* and *LinkUp*.
- *groupName*—A MIB conformance group name, such as *FixedLength* and *Objects*.

Cisco MIB Template

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 Cisco Template

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
    LAST-UPDATED      "9506130000Z"
    ORGANIZATION      "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."
REVISION      "9506130000Z"
DESCRIPTION
    "Initial version of this MIB module."
::= { ciscoMgmt 38 }
```

```
ciscoConfigManMIBObjects OBJECT IDENTIFIER ::= { ciscoConfigManMIB 1 }

ccmHistory          OBJECT IDENTIFIER ::= { ciscoConfigManMIBObjects 1 }

-- Textual Conventions

HistoryEventMedium ::= TEXTUAL-CONVENTION
    STATUS      current
    DESCRIPTION
        "The source or destination of a configuration change, save,
or copy.

        erase            erasing destination (source only)
        running         live operational data
        commandSource   the command source itself
        startup         what the system will use next reboot
        local           local NVRAM or flash
        networkTftp     network host via Trivial File Transfer
        networkRcp      network 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
    SYNTAX      TimeTicks
    MAX-ACCESS read-only
    STATUS      current
    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
    SYNTAX      TimeTicks
    MAX-ACCESS read-only
    STATUS      current
    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 }
ciscoConfigManMIBGroups      OBJECT IDENTIFIER ::= {
ciscoConfigManMIBConformance 2 }

-- Compliance

ciscoConfigManMIBCompliance MODULE-COMPLIANCE
    STATUS current
    DESCRIPTION
        "The compliance statement for entities which implement
         the Cisco Configuration Management MIB"
    MODULE      -- 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.

28.5.7 MIB Compilers

28.5.7.1 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.

28.5.7.2 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 `smicug.txt` file.

The SNMP Infra Tool at <http://mibportal:8080/SNMPInfraTool/index.html> includes the `smilint` compiler from `libsmi`. For more information, refer to the *SNMP Infra Tool User Guide*.

Another commonly used MIB compiler is `mosy`, which is part of the ISODE package. Although `mosy` is available at Cisco, it is not as complete as SMICng.

28.5.7.3 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 1** Login to a SunOS4.x or Solaris 2.x system.
- Step 2** Add the directory `/nfs/csc/mib-release/bin` to your shell's path variable.
- Step 3** If you have not done so already, create a MIB workspace:
`mibco.pl [release]`
- Step 4** Copy your new or modified MIBs into the `text-mibs` directory in your workspace.
- Step 5** Change directories into the root of your workspace.
- Step 6** 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
```

28.5.8 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 IOS Embedded Management Group.

28.5.9 Cisco Internal MIB Design Support

Individual technology groups typically design and implement their own MIBs. The IOS Embedded Management 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 `mib-police` email alias.

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 `cana` email alias.

28.6 MIB Review

The MIB Review is a *two tier* review process. This section provides an overview of the MIB Review, followed by subsections with more details on each part of the MIB Review process.

28.6.1 Overview of the MIB Review Process

A MIB author submits a MIB for review and approval. The MIB is subjected to compilation and sanity tests at the time of submission via the SNMP Infra Tool available at <http://mibportal:8080/SNMPInfraTool/index.html>. When the tests pass, the MIB is accepted into the MIB police system for the two tier review process and the MIB is assigned a reviewer and an approver.

28.6.1.1 Tier One

The reviewer reviews the MIB and gives comments to the author. The author, in turn, revises the MIB based on the comments and resubmits the MIB. The cycle is repeated until the reviewer is satisfied with the MIB and the MIB is given "reviewer approved" status.

28.6.1.2 Tier Two

Then, the approver reviews the MIB and can provide comments that cause the MIB to be revised by the author. The review cycle is similar to that with the reviewer, and finally, when the approver is satisfied with the MIB it is given "approved" status.

The system also has to get a unique OID (MIB number) assigned through CANA when a new MIB is approved.

28.6.1.3 Type of Review

The MIB police completes a design review. The MIB police reviewer and approver checks the following aspects of the MIB:

- 1 Does the overall structure of the MIB and naming conventions meet MIB police criteria?
- 2 Can any existing MIBs that are already defined in Cisco or in IETF be reused in place of this MIB?
- 3 Does the documentation in the MIB define all semantics as required? Can NMS developers read and understand the MIB, and develop an application to support this MIB?
- 4 Does the MIB meet the Cisco MIB standards?

Following the MIB design guidelines while designing a MIB will save time spent in revising the MIBs during a review and have a faster review turnaround.

28.6.2 MIB Submission Process

MIBs must be submitted to the MIB police system for review using the SNMP Infra Tool at <http://mibportal:8080/SNMPInfraTool/index.html>. Before submission, please make sure that the following things are taken care of:

- 1 The MIB is defined as per Cisco standard MIB template.
- 2 The MIB is written as per Cisco standard conventions.
- 3 Compilation check: MIB compiles using both `smilint` and `smicng`.

Once the submission passes through, every MIB is assigned a unique version number. For example:

`CISCO-TEST-MIB.my.1.0`

A re-submission of the MIB in a given review cycle will only change the minor version of a MIB; whereas, a resubmission after approval will change the major version.

28.6.3 Reviewer and Approver Assignment

The MIB reviewer and approver is selected based either on the submitter's choice or on the load factor (when auto-assign is chosen):

- A MIB submitter can choose the reviewer or approver during MIB submission.
- A MIB submitter can choose auto-assign. In this case, MIB reviewers and approvers are chosen based on the number of MIBs already assigned to them. The incoming MIB goes to the reviewer/approver with the least number of assignments. MIB reviewers/approvers can de-activate themselves from future assignments. The MIBs currently on their queue will continue to remain with them for review.

Once the reviewer and approver is selected, email that asks if they want to accept the MIB is sent to the reviewer and approver.

28.6.4 Approval Process

When the reviewer completes the review and is satisfied with the MIB, the MIB is approved and moved to "reviewer approved" state. The MIB is then marked as "action required from approver".

When the approver completes the review and is satisfied with the MIB, the MIB is approved and moved to "Approved" state.

Any state change triggers an email notification to the submitter, the interest-group, and the `mib-police` alias for archival.

When a new MIB (version 1.x) gets approved by the Cisco MIB police, a CANA number is automatically assigned to it.

28.6.5 Process Exceptions

Here are the known exceptions to the process:

- 1 Fast-track: MIBs on fast track are assigned only approvers (single pass review). To qualify for fast track, the submission must meet the following criteria:

- (a) Minor modification to an existing MIB
 - (b) Cisco-ized submission of an internet (IETF) draft
 - (c) Cisco-ized acquisition MIBs
 - (d) Non-Cisco-ized acquisition MIBs
- 2** Capability-MIBs: Capability MIBs are assigned only a reviewer and the approval from the reviewer is final.
- 3** CANA MIBs: These are MIBs that are originally approved by MIB police and require subsequent and frequent updates because of new OID assignments. These changes are exempted from MIB police. Currently, the CANA MIBs are:
- OLD-CISCO-CHASSIS-MIB
 - CISCO-PRODUCTS-MIB
 - CISCO-ENTITY-VENDORTYPE-OID-MIB

28.6.6 Policy on Acquisition MIBs

The acquisition's MIBs must be sent to `mib-police`. Since it is an acquisition, the `mib-police` members understand that the MIBs may already have been implemented and may be ready for shipping. The `mib-police` members will do the following:

- Step 1** If the acquisition wants to ship the MIB as-is, `mib-police` will require that the MIBs ship under the acquisition's existing OID space and a Cisco-oid will not be given for the MIB.
- Step 2** The MIB *must* compile with SMICng. If there are errors/warnings from SMIC and they cannot be fixed (for example, fixing will impact code), then the mib will *not* be published on CCO and other methods of publishing the field must be used.
- Step 3** `mib-police` may suggest minor corrections to the MIB, so long as the changes do not affect any implemented code.
- Step 4** After completing the previous 3 steps, the MIB will be published to CCO once. Any subsequent attempt to add new objects to the published MIB will likely be rejected.
- Step 5** The acquisition MIB is expected to be replaced with a new MIB that meets `mib-police` requirements.

28.6.7 Rejected, Withdrawn, and Dormant MIBs

If there are major design flaws in the MIB that make it unfit for review or another MIB satisfies the manageability requirements or Cisco has no authority on the MIB (for instance an IETF standard MIB), `mib-police` can reject the MIB. A submitter can withdraw a MIB from the MIB police review cycle (for instance, when the project is no longer pursued). If there is no activity by the submitter on the MIB for more than one month after a review comment is sent from `mib-police`, the MIB will be moved to the "dormant" state.

28.7 MIB Development Process: Overview

To develop a MIB, you perform the following tasks:

- Establish a New MIB

- Compile a MIB
- Observe Modularity
- Implement MIB Objects
- Implement SNMP Asynchronous Notifications
- Test a MIB
- Release a MIB
- Maintain a MIB

28.8 Establish a New MIB

To establish a new MIB, follow these steps. These steps use the sample MIB *BOOJUM-MIB.my*.

Step 1 Create a development tree.

Step 2 Do a **make depend** or a **make dependancies**) from the `sys` directory. This make generates code for all previously defined MIBs.

Step 3 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 Step 7 to end up.

Assume that the MIB for `boojum` support is `BOOJUM-MIB.my`. 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 `boojummib`. Also, we need to find the top-level identifier in `BOOJUM-MIB.my`. Let's assume that it is `boojumObjects`.

Step 4 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 5 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 Step 3. (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 6 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: BOOJUM-MIB.def
```

Step 7 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 8 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 files generated in Step 7. Therefore, you should not place these files under ClearCase control.

Step 9 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 10 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 11 Build the system and test the code.

Step 12 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 Step 10, and if applicable, the `boojum/sr_boojummibuser.h` and `sr_boojummib-mib.h` files.

28.9 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

28.9.1 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.

28.9.2 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.

28.9.3 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`.

28.9.4 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.

28.9.5 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. Table 28-1 explains the options. (Table 28-2 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 28-1 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 <code>sr_idtype.h</code> 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 <code>sr_iduser.h</code> file.
-snmpmibh	Enables the -snmpmibh switch in postmosy. When used in conjunction with the -codegen option, the -snmpmibh option generates an <code>sr_id-mib.h</code> file, which contains OID-to-textual-identifier mappings.
<i>mibs</i>	MIB definitions to be compiled.

Table 28-2 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.

28.9.6 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.

28.9.7 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_idmib.cfg` when you invoked the `mibcomp.perl` script.

- `../my_directory/sr_idmiboid.c`
- `../my_directory/sr_idmibdefs.h`
- `../my_directory/sr_idmibpart.h`
- `../my_directory/sr_idmibsupp.h`
- `../my_directory/sr_idmibtype.h`
- `../my_directory/sr_idmib.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_idmib.c`
- `../my_directory/sr_idmib.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`

28.9.8 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.my`. 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.my. 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.my. The CISCO-TCP-MIB.my file contains the following definition:

```
ciscoTcpConnEntry OBJECT-TYPE
SYNTAX      CiscoTcpConnEntry
MAX-ACCESS  not-accessible
STATUS      current
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.

28.10 Observe Modularity

You must observe modularity rules when dealing with SNMP subsystems and instrumentation.

28.10.1 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.

28.10.2 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.

28.11 Implement MIB Objects

This section discusses the following tips for creating SNMPv2 MIB method routines:

- GCC Warnings
- Validation
- k_get Routines
- k_set Routines

28.11.1 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.

28.11.2 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 setting 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 */
```

28.11.3 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.

28.11.4 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.

28.12 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

28.12.1 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.

28.12.2 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_configmamib.c
snmp/config_history.c
```

28.12.3 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 1 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 2 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 3 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 4 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_nvwr (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_nvwr(snark_trap_cfg_nvwr, "snark_trap_cfg_nvwr");
```

- Step 5** 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.

28.12.4 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 1** 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 2** 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 3** Register the service in the related MIB module:

```
reg_add_boojum(boojum, "boojum");
```

- Step 4** 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 1** 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 2** 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 3** Ensure that the family is enabled:

```
if (!boojum_traps_enabled)
    return;
```

Step 4 Build an instance vector:

```
instance[0] = index;
instanceOID.oid_ptr = instance;
instanceOID.length = 1;
```

Step 5 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 6 Have SNMP generate the trap:

```
snmp_trap(ENTERPRISE_TRAP, BOOJUM_NUMBER, vbList, (OID *)&enterpriseOID,
boojumOID_str);
```

Step 7 Clean up after CatOID:

```
for (i = 0; i < TRAP_VARBIND_COUNT; i++) {
    FreeOID(vbList[i]);
}
```

28.13 Test a MIB

When testing a MIB, you check the following basic SNMP operations on objects and notifications:

- Get
- GetNext
- Set
- Trap

28.13.1 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.

28.13.2 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.

28.13.3 Tools for Testing a MIB

To test a MIB, you can use command-line tools, X Windows tools, and notification tools.

28.13.3.1 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 1** Create a directory.
- Step 2** Copy all the files from `/nfs/csc/snmpv2/mibs` into the directory you created.
- Step 3** In the new directory, change the `makefile` as follows:
- Add your new MIB.
- Step 4** Do a `make` in that directory.
- Step 5** Set the environment variable `SR_MGR_CONF_DIR` to point at the new directory.

28.13.3.2 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 it on a nonmail server that has spare CPU cycles.

To run `xsnmptcl` the test suite, follow these steps:

- Step 1** Change into the `xsnmptcl` directory:
- ```
cd /atml/kzm/isode-8.0/snmpV2/tcl/library
```
- Step 2** Start `xsnmptcl` for testing:
- ```
./xsnmptcl -f testing.tcl
```

The main window appears .

Step 3 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 4 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  
/atml1/kzm/isode-8.0/snmpV2/mosy/xmosy yourmib.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.

28.13.3.3 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 SPARC as 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
```

28.13.4 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.
- Create and delete first, intermediate, last, and overflow entry.

28.13.5 Object Functions

Confirm that each object works as specified in the MIB:

- Get returns an operationally correct value.
- Set has an operationally correct effect.

28.14 Release a MIB

28.14.1 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 `mib-police` email alias.

28.14.2 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 `mib-release` email alias.

28.15 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_xxxmib.c` code that was previously created. In this case, the `sr_xxxmib.stamp` file contains the skeleton code for the new MIB. It is up to you to compare the existing `sr_xxxmib.c` code to the new skeleton code in `sr_xxxmib.stamp` and to make all appropriate changes to the `.c` file before committing the MIB (and the `.c` file) to the release tree.

28.15.1 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 new `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 1** Add a `REVISION` clause to the `MODULE-IDENTITY` macro explaining the update.
- Step 2** Update the `LAST-UPDATED` clause in the `MODULE-IDENTITY` macro.

- Step 3** 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 4** 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 5** 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 6** 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 1** Update MODULE-IDENTITY.

- Step 2** 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 3** Create a new OBJECT-GROUP macro that does not contain the deleted objects.

- Step 4** Create a new MODULE-COMPLIANCE macro that references the new object group macro.

28.16 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.

28.16.1 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 Cisco IOS 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-Cisco IOS MIBs. That is, the MIB files you are testing do not already have to exist in a Cisco IOS source repository.

To create or update a MIB workspace, follow these steps:

Step 1 Locate a disk partition that has at least 25-30 MB of free space.

Step 2 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 Cisco IOS 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 Cisco IOS version.

Step 3 Change into the directory in which you want the workspace to be created.

Step 4 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.

28.16.2 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 Cisco IOS 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 1 Log in to a system that meets the following criteria:

- It is running SunOS 4.x or SunOS 5.x (also known 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 2 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 3 If you have not already created a MIB workspace, as described in the section "Create or Update a MIB Workspace," do so now.

Step 4 Copy your new or modified MIB (the `*.my` files) into the `test-mibs` directory in your workspace.

Step 5 Change your current directory to the root of your workspace.

Step 6 If you are testing changes to an SNMPv1 MIB, issue the following command:

```
update-mibs.pl -1
```

Step 7 If you are testing changes to an SNMPv2 MIB, issue the following command:

```
update-mibs.pl
```

28.16.3 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.my`, 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.

28.16.4 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 `mib-release` email alias.

28.16.5 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 Cisco IOS release, the MIB release group creates the SNMPv1 versions and provides them to the customers. If your MIB is a non-Cisco IOS 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.”

28.16.6 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

28.16.7 Use Make Directly to Generate an SNMPv2 MIB

To use the `make` command directly to generate an SNMPv2 MIB, follow these steps:

Step 1 Change into the root directory in your MIB workspace.

Step 2 Change into the `v2` directory:

cd v2

Step 3 Test the SNMPv2 MIB to ensure that its syntax is correct:

make depend

make all

28.16.8 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 1 Change into the root directory in your MIB workspace.

Step 2 Convert the SNMPv2 MIBs into SNMPv1 MIBs:

make depend

Step 3 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.my`.

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.

28.16.8.1 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.my`.

28.16.9 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 `cco-team` email alias.

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/betaxxx_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`.

28.16.10 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 `mib-release` email alias. 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.

28.16.11 MIB-Related Files

28.16.12 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`.

28.16.13 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.

28.16.14 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.

28.16.15 Directory Layout for MIB Repository and Workspace

A MIB repository or workspace has the following directory structure. Table 28-3 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 28-3 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.my.
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.
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 28-4 explains the files related to the `makefiles`.

Table 28-4 makefile Structure

File/Directory	Contents
makefile.defs	Common definitions used by all <code>makefiles</code> , for example, program names, locations, and options, and lists of MIB files to process or ignore. This file is included by all <code>makefiles</code> in the tree.
makefile.common	Rules common to all <code>makefiles</code> in the tree. This file is included by all the <code>makefiles</code> .
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 <code>makefiles</code> 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.

MIB Infrastructure

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.

Note Cisco IOS MIB infrastructure questions can be directed to the mib-dev@cisco.com and mib-police@cisco.com aliases.

29.1 Overview

This chapter includes information on the following topics:

- Infrastructures provided by certain core MIBs to be used by other subsystems as needed.
 - Interface Manager Infrastructure for IF-MIB, CISCO-IF-EXTENSION-MIB, and interface-specific MIBs (starting with release 12.2SRA and 12.2SB)
 - IF-MIB Infrastructure
 - Entity MIB Infrastructure
- Infrastructures provided by SNMP and associated subsystems to be used by MIBs.
 - MIB Persistence Infrastructure
 - SNMP Notification Infrastructure
 - HA MIB Sync Infrastructure
- References

29.1.1 Terminology

GSR

Gigabit Switch Router

Interface Stack

An abstraction of the layering between a set of interfaces.

Interface Sub-stack

A given interface and zero or more sub-layer interfaces.

HA

High Availability

IF-MIB

Interfaces Group MIB (Management Information Base). The current specification for the IF-MIB is found in RFC 2233. The MIB module to describe generic objects for network interface sublayers. This MIB is an updated version of the MIB-II if Table, and incorporates the extensions defined in RFC 1229.

IFS

IOS File System

ILMI

Integrated Local Management Interface

MIB

Management Information Base

NM

Network Management

NMS

Network Management Station

NVRAM

Non-Volatile Random Access Memory

OIR

Online Insertion Removal. The process responsible for monitoring the insertion and removal of modules during the nominal operation of a system; hot swap.

Physical Layer Interface Module (interface)

Physical Layer Interface Module: any module (for example, field replaceable unit) that supports one or more physical network interfaces, such as a native line card, port adapter, or shared port adapter.

RF

Redundancy Facility, provided by IOS HA infrastructure to notify its clients of active and standby state progression events.

Sub-Interface

A communication channel created within an interface capable of transmitting and/or receiving IP packets. With respect to a given interface, a sub-interface will always appear immediately above that interface in the interface stack.

Sub-Layer Interface

See Subordinate Interface.

Subordinate Interface

An interface appearing below a given interface in an interface stack.

Superior Interface

An interface appearing above a given interface in an interface stack.

USM

User-based Security Model

29.2 Interface MIBs Infrastructure Overview

There are two interface MIB infrastructures in Cisco IOS: the IF-MIB infrastructure (prior to releases 12.2SRA and 12.2SB), and the new Interface Manager infrastructure which is intended to replace the IF-MIB infrastructure.

29.2.1 Interface Manager Infrastructure

This section was added August 2006.

The IM is a new infrastructure that replaces the existing IF-MIB infrastructure and provides the building blocks for the development of interface-specific MIBs (for example, SONET-MIB). It simplifies management of interfaces by acting as a “common control point” between management clients (subsystems managing data for a specific interface driver) and interface drivers (subsystems owning specific interface driver data). This common control point allows any client that requires configuring, updating, monitoring or diagnosing a specific interface, to access the data in a consistent way. It allows interface drivers to make available (to the clients) their data in a consistent way without having to expose their internal data structures.

29.2.1.1 Background

Since the first Cisco devices were created, there has always been the need to manage them. Whether the device management is done via a local management (CLI) or by remote management (that is, SNMP, XML) the data being retrieved or modified must be accessed in a consistent way.

Unfortunately, this is not the case in most Cisco devices. The network management of interfaces has become increasingly problematic on systems running IOS and affecting many phases of a product’s life-cycle:

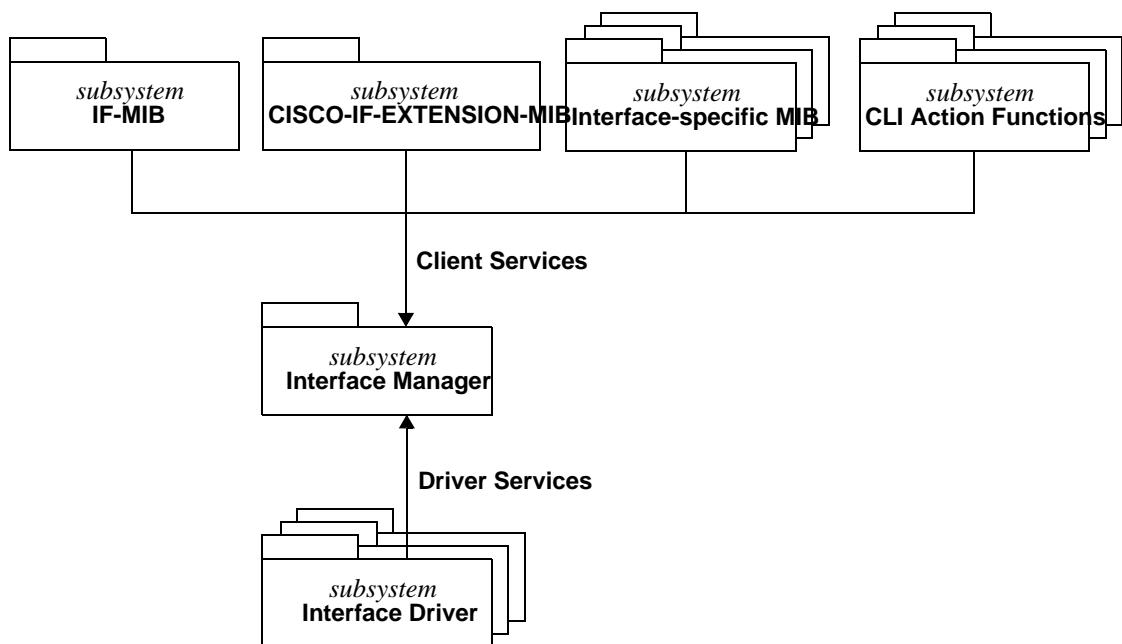
- Development—the code-base supporting the IF-MIB and interface-specific MIBs does not ease an interface driver from having knowledge specific to network management. For example, the burden of creating and destroying conceptual rows in the ifTable and ifStackTable largely falls on interface drivers. Many driver developers do not have experience with network management, and this task becomes onerous to the overall effort. As result, a project team typically prioritizes the support of these MIBs so low that they either forego the task, or assign resources inadequate to guarantee acceptable quality.
- Test—the code-base supporting the IF-MIB and interface-specific MIBs does not expose hooks lending to timely and cost effective compliance and functional testing. This fact coupled with the low priority typically assigned to these features by the project team has resulted in a negligible investment in developing comprehensive compliance and functional tests for these MIBs. The attitude toward testing these features has become so entrenched that it has become typical for a interface to be shipped without these MIBs ever being tested, which further exacerbates the aforementioned quality problem.
- Maintenance—the code-base supporting the IF-MIB and interface-specified MIBs has become contaminated with numerous platform-dependent and interface-dependent references. This problem has made it increasingly difficult to migrate to new revisions of these MIBs that have come out over the years. In addition, the problem has made it difficult to isolate issues reported by the field, which increases the time and resources required to fix these types of issues.

From the perspective of our customers, these problems manifest themselves in the form of conformance and consistency issues. These issues can significantly increase the cost of developing applications for the purpose of managing our IOS-based devices. In addition, these issues translate into support costs for Cisco.

29.2.1.2 Interface Manager

By acting as a common control point, the IM exposes a set of client services to the management clients, such as CLI and MIB subsystems, for the purpose of retrieving and modifying managed data associated with interfaces. It also exposes a set of driver services to the interface drivers for the purpose of creating/destroying interfaces, retrieval of managed data, validation of configuration data, modification of configuration data, posting status, and signaling of events and alarms. Figure 29-1 shows a black box view on how the IM acts as a common control point between clients and drivers.

Figure 29-1 Interface Manager black-box view

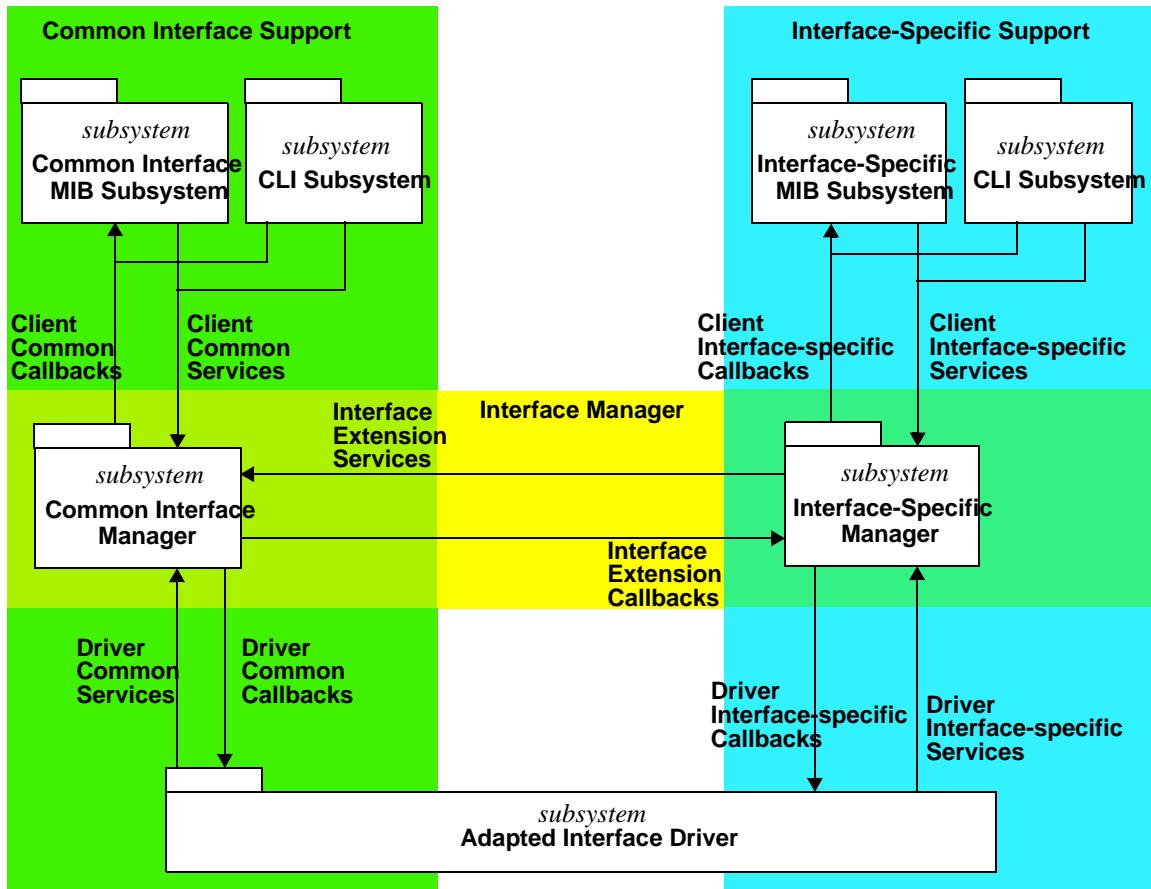


It's important to note that in this figure, clients are not limited to just MIB or CLI subsystems. These clients can be any type of subsystem (for example, XML and CLI prototype) that requires the access of the data in the driver. Same applies to the driver subsystems. For example, a developer can now create a pseudo driver that looks and acts like a driver for the purpose of unit testing a client.

29.2.1.2.1 IM Architecture

The IM consists of the Common Interface Manager (CIM) subsystem and one or more Interface-Specific Manager (ISM) subsystems. The CIM is the main subsystem that provides the framework to manage all interfaces. An ISM can be viewed as an extension of the CIM for the purpose of managing a specific type of interface. For example, the SONET-MIB subsystem would be classified as an interface-specific subsystem, because it only supports interfaces having the ifType of 'sonet', 'sonetPath', and 'sonetVT'. These subsystems interact with the CIM through well-defined interfaces, and forbid any given subsystem from directly accessing data maintained by another subsystem.

Figure 29-2 IM Architecture



Both the CIM and ISM subsystems provide two types of APIs defined in registries: services and callbacks. A service is a function defined by a subsystem and exposed to other subsystems. For example, a service in IOS is a service point contained by a registry. A callback is a function defined by a subsystem and volunteered to another subsystem through a service exposed by that subsystem. For example, a callback in IOS is a function registered with a LIST, LOOPBACK, or CASE service point using `reg_add_xxx()`.

29.2.1.2.2 IM Concepts

Interface Signature

An interface signature is an enumerated value that uniquely identifies the type of interface sub-stack being created. These interface signatures should not be confused with the IANAifType textual convention defined by IANAifType-MIB, which is published by the IANA.

Interface Handle

The interface handle is an opaque 32-bit value that uniquely identifies the interface being created. This value will typically be a pointer to the driver-specific construct representing the interface (for example HWIDB, SWIDB and subblock).

Driver Identifier

The driver identifier is an 8-bit value that uniquely identifies the driver to which an interface handle belongs. This driver identifier is obtained by registering the driver with the IM during the driver's subsystem initialization.

Stack Management

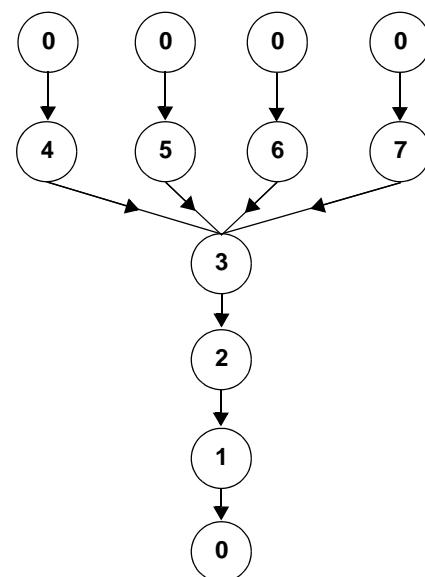
An interface stack describes the layering of a set of related interfaces. Figure 29-3 illustrates an example of an interface stack. With respect to any given interface in an interface stack, we refer to an interface located immediately above that interface as a superior interface. Likewise, we refer to an interface located immediately below that interface as a subordinate interface.

Figure 29-3 Interface Stack

[4] 'frDlciEndpt'	[5] 'frDlciEndpt'	[6] 'frDlciEndpt'	[7] 'frDlciEndpt'
[3] 'frameRelay'			
[2] 'sonetPath'			
[1] 'sonet'			

An interface stack can be represented by a directed, non-cyclic graph of interfaces. In such a graph, each node represents an interface, while each connection represents a relationship between two interfaces. Figure 29-4 illustrates such a graph representing the interface stack depicted in Figure 29-3. If an interface has no superior interfaces, then we add a null interface on top of it. If an interface has no subordinate interfaces, then we add a null interface below it.

Figure 29-4 Graph Representing Interface Stack



The IF-MIB defines the ifStackTable, which describes the interface stacks within the scope of a SNMP entity. Each conceptual row in this table describes a connection in a graph representing an interface stack. The table identifies a connection by the ifIndex associated with the superior and subordinate interface, in the order specified. Thus, a connection flows from superior to subordinate interface (as illustrated in Figure 29-4).

Interface Index

The interface index is the ifIndex-value assigned to a particular entry in the ifTable.

Interface Parent Index

The interface parent index is the ifIndex value assigned to subordinate interface being created.

Master Interface Index

The master interface index is the ifIndex-value assigned to the interface used for deriving statistics and sub-layer interface representation.

Legacy Interface

A legacy interface is an interface maintained by an interface driver that has not been modified to employ the IM framework.

Interface Driver

A subsystem responsible for providing a layer of abstraction between device drivers or other data path components (for example, SSS) and the IOS operating environment.

29.2.1.2.3 Type Definitions

The `/vob/ios/sys/if_mgr/im_public_types.h` file has definitions for data types that are needed by interface drivers to invoke its services. For example, this file has definitions for interface signature enumerations (`im_if_signature_t`) and interface specification data structure (`im_if_spec_t`).

29.2.1.2.4 Driver Registry

The `/vob/ios/sys/if_mgr/im_driver_registry.reg` file contains the definitions for the services provided by CIM to interface drivers (for example, POS SPA interface driver). Also, this registry contains definitions for driver callbacks. The CIM adds to the following services:

- `im_driver_register` —Allows interface drivers to register callbacks required by the IM to manage data relating to a specified type of interface being maintained by the driver. A driver must invoke this service before any other interaction with the IM relating to interfaces of the specified type.
- `im_driver_deregister` —Allows interface drivers to revoke the callbacks relating to a specified type of interface previously registered with the IM. The interface driver MUST explicitly destroy all interfaces of the specified type previously registered with the IM before invoking this service. After invoking this service, there should be no further interaction between the IM and the driver relating to interfaces of the specified type until the driver invokes the `im_driver_register` service again.

- `im_driver_if_create` —Allows an interface driver to register an interface with the IM. A driver should call this service after it has created an instance of an object representing an interface. Furthermore, if the interface being registered has a subordinate interface, then the subordinate interface must previously have been created.
- `im_driver_if_destroy` —Allows an interface driver to deregister an interface with the IM. A driver should call this service before it destroys an instance of an object representing an interface. Furthermore, an interface should only be destroyed if it is on the top of an interface stack.
- `im_driver_if_stack_relationship_add` —Allows interface drivers to create a connection between two interfaces in an existing interface stack.
- `im_driver_if_stack_relationship_delete` —Allows interface drivers to destroy a connection between two interfaces in an existing interface stack.
- `im_driver_if_stack_relationship_get` —Allows an interface driver to retrieve managed data relating to a connection between interfaces in an interface stack. This service identifies a connection uniquely by a pair of ifIndex values identifying the superior and subordinate interfaces (in that particular order) making up the connection.
- `im_driver_if_rcv_address_present` —Allow an interface driver to indicate if it is maintaining receive addresses for a given interface index. When the driver maintains 1 or more receive addresses for a given index, the `rcvAddrPresent` flag should be set to TRUE. When the driver is not maintaining any receive addresses for a given interface, the `rcvAddrPresent` flag should be set to FALSE.
- `im_driver_event_generate` —This service allows a driver to generate an interface event.

The interface drivers add to the following services:

- `im_driver_if_data_get_callback` —Allows interface drivers to return managed data relating to a particular interface.
- `im_driver_if_data_validate_callback` —Allows interface drivers to validate configuration data relating to a particular interface.
- `im_driver_if_data_set_callback` —Allows interface drivers to commit/set configuration data relating to a particular interface.
- `im_driver_if_rcvaddr_enumerate_callback` —Provided by drivers to enumerate receive addresses for a given interface and enumeration index. The enumeration index should start at 0 for to enumerate the first receive address for an interface.

29.2.1.3 Interface Driver

An interface driver is any subsystem maintaining the notion of an interface, whether the interface is physical, logical, or virtual in nature.

- Register with the CIM to obtain a driver identifier by invoking the driver registration service in `im_driver_registry.reg`.
- Invoke the interface creation/deletion services in `im_driver_registry.reg` to create or delete the appropriate layers corresponding to a specific interface.
- Implement the `im_driver_registry.reg` callbacks to get/set/validate specific management data maintained by the driver.

29.2.1.3.1 Interface Driver Initialization

Before any interface driver can make use of the CIM services, it must obtain a unique driver identifier. Interface drivers can obtain a driver identifier by invoking the `im_driver_register()` service during the driver's initialization. Note that, if the device supports multiple interface drivers with the same interface signature, then each driver subsystem with the same signature must individually register with the CIM and obtain a driver identifier. For example, the Cisco 7600 supports multiple line card technologies, such as OSM, PA, SPA, for the same type of interface (for example, POS and SONET). Each of these line cards has their own interface driver subsystem, therefore, each of these interface driver must register and obtain a driver identifier from the CIM.

Once a driver identifier is obtained, the interface driver must implement the appropriate callback functions that it supports by invoking the `im_driver_if_data_xxx_callback()` services.

Example 1 (Ethernet Driver)

The following example shows a simple registration of an Ethernet driver supporting the `IM_ETHERNET` signature:

```
#include "if_mgr/im_public_types.h"

/*
 * A globally unique identifier assigned to this driver by the IM.
 */
static im_driver_id_t driverID;

ether_subsys_init () {
/*
 * First, register with the IM to obtain a driver ID
 */
    reg_invoke_im_driver_register(IM_ETHERNET, &driverID, &status);
    if (status == IM_PASS) {
    /*
     * If driverID was obtained, then register with IM the driver's callback
     * functions
     */
        (void) reg_add_im_driver_if_data_get_callback(IM_ETHERNET,
            xx_if_get_data, "xx_if_get_data");
        (void) reg_add_im_driver_if_data_validate_callback(IM_ETHERNET,
            xx_if_validate_data, "xx_if_validate_data");
        (void) reg_add_im_driver_if_data_set_callback(IM_ETHERNET,
            xx_if_set_data, "xx_if_set_data");

    /* Do the callback registration for RcvAddrTable, if receive address is
     * maintained by this driver */
        (void) reg_add_im_driver_if_rcvaddr_enumerate_callback(IM_ETHERNET,
            xx_if_rcvaddr_enumerate_callback,
            "xx_if_rcvaddr_enumerate_callback");
    } else {
    /*
     * Print Warning Error Message
     */
    }
}
```

Example 2 (VLAN Driver)

The following example shows a subsystem that can handle more than one type of signature. In this case, we will use a VLAN driver as an example, which can support both DOT1Q and ISL signatures:

```
#include "if_mgr/im_public_types.h"

/*
 * A globally unique identifier assigned to this driver by the IM.
 */
static im_driver_id_t dot1q_vlan_driverID;
static im_driver_id_t isl_vlan_driverID;

vlan_subsys_init () {
/*
 * First, register the VLAN DOT1Q with the IM to obtain a driver ID
 */
    reg_invoke_im_driver_register(IM_VLAN_DOT1Q, &dot1q_vlan_driverID,
&status);
    if (status == IM_PASS) {
        /* If driverID was obtained, then register with IM the driver's callback
functions */
        (void) reg_add_im_driver_if_data_get_callback(IM_VLAN_DOT1Q,
vlan_if_get_data, "vlan_if_get_data");
        (void) reg_add_im_driver_if_data_validate_callback(IM_VLAN_DOT1Q,
vlan_if_validate_data, "vlan_if_validate_data");
        (void) reg_add_im_driver_if_data_set_callback(IM_VLAN_DOT1Q,
vlan_if_set_data, "vlan_if_set_data");
    } else {
        /*
         * Print Warning Error Message
         */
    }
/*
 * Second, register the VLAN ISL with the IM to obtain a driver ID
*/
    reg_invoke_im_driver_register(IM_VLAN_ISL, &isl_vlan_driverID, &status);
    if (status == IM_PASS) {
        /*
         * If driverID was obtained, then register with IM the driver's
         * callback functions
         */
        (void) reg_add_im_driver_if_data_get_callback(IM_VLAN_ISL,
vlan_if_get_data, "vlan_if_get_data");
        (void) reg_add_im_driver_if_data_validate_callback(IM_VLAN_ISL,
vlan_if_validate_data, "vlan_if_validate_data");
        (void) reg_add_im_driver_if_data_set_callback(IM_VLAN_ISL,
vlan_if_set_data, "vlan_if_set_data");
    } else {
        /*
         * Print Warning Error Message
         */
    }
}
```

29.2.1.3.2 Interface Creation

Interface creation always originates from an interface driver; therefore the following assumptions are supported:

- The boot process invokes an interface driver to create a static interface corresponding to physical ports supported by the system.
- Upon detecting the insertion of an interface, OIR invokes the interface driver to create static interface(s) that correspond to physical port(s) supported by the interface.
- Several commands supported by the CLI invoke the interface driver to create one or more static interfaces known as sub-interfaces, such as channelized interfaces, Ethernet VLANs, and ATM VCs.
- Various control plane components, such as the Subscriber Service Switch (SSS)¹ have the capability of creating dynamic interfaces (for example, as a result of PPP session establishment). In this case, the IM treats the control plane component as an “interface driver.”

From the perspective of the CIM, and given the above assumptions, interface creation starts with an interface driver invoking the `im_driver_if_create()` service. The interface driver should invoke this API whenever it has created or initialized an object representing an interface (for example, HWIDB, SWIDB and sub-block). It's important to note that a driver doesn't necessarily need to have a HWIDB or SWIDB to create an interface. This is a misconception that has previously led to numerous missing layers in the ifStack table.

When the interface invokes the `im_driver_if_create()` service, it must provide an interface specification which contains the following fields:

- Interface Signature (`ifSignature`) — an enumerated value that uniquely identifies the type of interface stack being created.
- Driver Identifier (`driverId`) — 8-bit value that uniquely identifies the driver to which the `ifHandle` belongs.
- Interface Handle (`ifHandle`) — an opaque, 32-bit value uniquely identifying the interface being created. This value will typically be a pointer to the driver-specific construct representing the interface.
- Interface Parent `ifIndex` (`ifParentIndex`) — `ifIndex` value assigned to the interface subordinate to the interface being created. If an interface does not have a parent interface, then the `ifParentIfIndex` must be set to 0. For example, when creating a POS interface, this may be the `ifIndex` assigned to the SONET path supporting the POS interface.
- Interface Name (`ifName`) — a string assigned to the interface by the local console.
- Interface Descriptor (`ifDescr`) — a string assigned to the interface by the system. This value is required to be globally unique and persistent across restarts.
- Master Interface (`ifIndex`) — the index of the interface that can be used to derive statistics for interface sub-layers created by the IMACP. When an interface driver creates an interface (for example, not created by the IMACP), this value will typically be the same as the `ifParentIndex`. See “29.2.1.2.2 IM Concepts” for more information regarding interface sub-layers, stacks, and master interfaces.

Once the driver has registered the interface with the CIM and obtained an `ifIndex` for the interface, the driver must indicate to the CIM (via the `im_driver_if_rcv_address_present()` service) whether or not the interface maintains any receive address.

Note Most IOS drivers make use of the `idb_final_hw_init()` function either directly or indirectly via some other function while initializing a new HWIDB. It is important that before calling the `idb_final_hw_init()`, the `hwidb->snmp_if_index` be set to other than zero. This can be done by creating the new interface with the `im_driver_if_create()` service and passing the `hwidb->snmp_if_index` as argument before calling the `idb_final_hw_init()`. If for whatever reason the `im_driver_if_create` service() fails to provide a valid `ifIndex`, then the driver code must display a warning syslog and set the `hwidb->snmp = MAXUINT` to prevent the legacy code from registering the new interface.

Note It's very important to create all interface layers in the order that they get initialized (from the bottom layer to the top layer). For example, if a POS line card is OIRed, then the driver would create the following layers:

1. Create SONET
2. Create SONET Path
3. POS

Failing to create all interface layers may inhibit a network management station from discovering all appropriate layers, and use the appropriate specific MIB for each layer type.

Example: Ethernet Interface

The following example uses a HWIDB construct to represent an Ethernet interface, but the same concept applies to any other type of construct (for example, SWIDB and subblock).

```
#include "if_mgr/im_public_types.h"

/*
 * Example of a function that will be called every time a hwidb. Notice, this
 * hwidb is only associated with
 * an Ethernet port.
 */
ether_if_init (hwidbtype *hwidb)
{
    im_if_spec_t ifSpec;
    im_status_t status;
    im_status_t imCompletionStatus = IM_FAIL;

    if (hwidb->snmp_if_index) {
        /* Already registered, just return */
        return;
    }

    /* If the namestring has not being initialized yet, do it now. */
    if (!hwidb->hw_namestring)
        idb_init_names(hwidb->firstsw, FALSE);

    ifSpec.ifSignature = IM_ETHERNET;

    /* Use the driver identifier previously obtained during the subsystem
     * initialization */
    ifSpec.driver_id = driverID;
```

```

ifSpec.if_handle = (im_if_handle_t )hwidb;

/* Since the Ethernet port is the physical port, there is no need to
provide a parent or master index */
ifSpec.if_parent_index = 0;
ifSpec.if_master_index = 0;
ifSpec.ifName = hwidb->hw_short_namestring;
ifSpec.ifDescr = hwidb->hw_namestring;
hwidb->firstsw->sw_status.snmp_link_notification = TRUE;

reg_invoke_im_driver_if_create(&ifSpec,&hwidb->snmp_if_index,
                               &status);
if (status != IM_PASS && status != IM_IF_PRESENT) {
/* Print Warning Syslog to indicate that the IM failed to add the specific
interface */
} else {
    /* Use TRUE if this driver maintains Receive Address in this
interface, else use FALSE */
    reg_invoke_im_driver_if_rcv_address_present(hwidb->snmp_if_index,
TRUE, &imCompletionStatus);
    if (imCompletionStatus == IM_FAIL) {
        /* Print Warning Syslog to indicate that the IM failed to set the
rcv address */
    }
}
}

```

29.2.1.3.3 Interface Destruction

The IM assumes that interface destruction always originates from the driver that created the interface. This assumption supports the following:

- Upon detecting the removal of an interface, OIR invokes the interface driver to destroy static interface(s) that correspond to the physical port(s) supported by the interface.
 - Several commands supported by the CLI invoke the interface driver to destroy one or more static interfaces known as sub-interfaces, such as channelized interfaces, Ethernet VLANs, and ATM VCs.
 - Several control plane components, such as the Subscriber Service Switch have the capability of destroying dynamic interfaces (for example, as a result of PPP session termination). In this case, the IM treats the control plane component as an “interface driver.”

From the perspective of the IM, and given the above assumptions, interface destruction starts with an interface driver invoking the `im_driver_if_destroy()` service. The interface driver should invoke this service whenever it has destroyed or freed an object representing an interface (for example, HWIDB, SWIDB, sub-block, etc.). The driver no longer has to worry about keeping track of the ifStack table since the CIM will take care of this process while removing interfaces from its table (See section “29.2.1.3.8 Interface Stack Relationships”, for exceptions to this rule).

Note It's very important to destroy all interface layers in the reverse order that they were created (from the top layer to bottom layer). For example, if a POS line card is OIRed, then the driver would have to delete the following layers before completely removing the POS HWIDB:

1. Delete POS layer
2. Delete SONET Path layer
3. Delete SONET layer

Failing to destroy all interfaces will cause a failure when the interface is recreated again.

Delete Example (Ethernet Interface)

The following example shows how a driver would remove a previously created Ethernet interface:

```
static void
ether_if_removed (slottype *sp)
{
    uint ix;
    hwidbtype *hwidb;
    im_status_t status;

    for (ix = 0; ix < sp->interface_count; ++ix) {
        hwidb = sp->interface_list[ix];
        if (hwidb) {
            /* De-register all entries from the ifTable */
            if (hwidb->snmp_if_index) {
                reg_invoke_im_driver_if_destroy(hwidb->snmp_if_index,
                                                hwidb->hw_short_namestring, &status);
                hwidb->snmp_if_index = 0;
                if (status != IM_PASS) {
                    /* Print Warning Syslog error */
                }
            }
        }
    }
}
```

29.2.1.3.4 Interface Get Data

The interface driver is responsible for implementing the `im_driver_if_data_get_callback()` service in order to make available specific interface data (for example, name, description, speed and status) to the CIM. It is important to note that `im_driver_if_data_get_callback()` is a CASE_LOOP service, which means that there can be one or more drivers registered with the same signature; therefore, each driver is responsible for making sure that the driver ID being passed correspond to them. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface (for example, HWIDB, SWIDB and subblock).

Using the `im_if_mgmt_data_t` data structure, the interface driver can determine which set of data the CIM is requesting, and which field(s) within the set of data it needs to retrieve. If one or more fields are not available or not supported by the driver interface, then the driver can clear the specific fields that it can support for the specific set of data.

The status field should be used by the interface driver to indicate any error while retrieving the specific data.

Get Data Example

This example shows a generic function to retrieve data from the driver:

```

static boolean
xx_if_get_data (im_driver_id_t id,
                im_if_handle_t ifHandle,
                im_if_mgmt_data_t *ifData,
                im_status_t *status)
{
    Handle type* handle; (e.g. hwidb, swidb, subblock, etc.)

    *status = IM_FAIL;

    /*
     * Check if this driveID correspond to this driver. The driverID was
     * previously obtained during the
     * subsystem init and should be global.
     */
    if (driverID != id) {
        return FALSE;
    }

    handle = (handle type*)handle;
    if (!handle) {
        return TRUE;

        /* Call the specific function to handle the request */
        if (ifData->im_if_data_type == IM_IF_GENERAL_STATS)
            *status = xx_if_get_gen_stats(handle,
&ifData->im_if_data.im_if_gen_stats);
        else if (ifData->im_if_data_type == IM_IF_PACKET_STATS)
            *status = xx_if_get_packet_stats(handle,
&ifData->im_if_data.im_if_packet_stats);
        else if (ifData->im_if_data_type == IM_IF_EXTENSION_STATS)
            *status = xx_if_get_ext_stats(handle,
&ifData->im_if_data.im_if_ext_stats);
        else if (ifData->im_if_data_type == IM_IF_UTILIZATION_STATS)
            *status = IM_FAIL; /* Not supported */
        else {
            return TRUE;
        }
    }
}

```

29.2.1.3.5 Interface Validate Data

The interface driver is responsible for implementing the `im_driver_if_data_validate_callback()` service to validate configuration specific interface data that it is about to be changed. It is important to note that `im_driver_if_data_validate_callback()` is a CASE_LOOP service, which means that one or more driver can be registered with the same signature; therefore, each driver is responsible for making sure that the driver ID being passed corresponds to them. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface (for example, HWIDB, SWIDB and subblock).

Using the `im_if_mgmt_data_t` data structure, the interface driver can determine which set of data the CIM is requesting, and which field(s) within the set of data it needs to validate. If one or more fields are not allowed to be changed or not supported by the driver interface, then the driver can clear the specific fields that it can support for the specific set of data.

The status field should be used by the interface driver to indicate any error while retrieving the specific data.

Validate Data Example

This example shows a generic function to validate data:

```
static boolean
xx_if_validate_data (im_driver_id_t id,
                     im_if_handle_t ifHandle,
                     im_if_mgmt_data_t *ifData,
                     im_status_t *status)
{
    hwidbtype *hwidb; (e.g. hwidb, swidb, subblock, etc.)

    *status = IM_FAIL;

    /*
     * Check if this driveID correspond to this driver. The driverID was
     * previously obtained during the
     * subsystem init and should be global.
     */
    if (driverID != id) {
        return FALSE;
    }

    hwidb = (hwidbtype*)ifHandle;
    if (!hwidb) {
        return TRUE;
    }

    /* Call the specific function to handle the request */
    if (ifData->im_if_data_type == IM_IF_GENERAL_STATS)
        *status = cwan_poseidon_validate_gen_stats(hwidb,
                                                &ifData->im_if_data.im_if_gen_stats);
    else if (ifData->im_if_data_type == IM_IF_PACKET_STATS)
        *status = IM_FAIL; /* not supported */
    else if (ifData->im_if_data_type == IM_IF_EXTENSION_STATS)
        *status = cwan_poseidon_validate_ext_stats(hwidb,
                                                &ifData->im_if_data.im_if_ext_stats);
    else if (ifData->im_if_data_type == IM_IF_UTILIZATION_STATS)
        *status = IM_FAIL; /* not supported */
    else {
        return TRUE;
    }
}
```

29.2.1.3.6 Interface Set Data

The interface driver is responsible for implementing the `im_driver_if_data_set_callback()` service to set a specific interface data (for example, admin Status and trap enable). It is important to note that `im_driver_if_data_set_callback()` is a CASE_LOOP service, which means that there can be one or more driver registered with the same signature; therefore, each driver is

responsible for making sure that the driver ID being passed correspond to them. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface (for example, HWIDB, SWIDB and subblock).

Using the `im_if_mgmt_data_t` data structure, the interface driver can determine which set of data the CIM is requesting, and which field(s) within the set of data it needs to set. If one or more fields are not available or it can not be modified by the driver interface, then the driver can clear the specific fields that it can support for the specific set of data.

The status field should be used by the interface driver to indicate any error while retrieving the specific data.

Set Data Example

This example shows a generic function to set data:

```
boolean
xx_if_set_data (im_driver_id_t id, im_if_handle_t ifHandle,
im_if_mgmt_data_t *ifData, im_status_t *status)
{
    hwidbtype *hwidb;

    *status = IM_FAIL;

    /*
     * Check if this driveID correspond to this driver. The driverID was
     * previously obtained during the subsystem init and should be
     * global.
     */
    if (driverID != id) {
        return FALSE;
    }

    hwidb = (hwidbtype*)ifHandle;
    if (!hwidb) {
        return TRUE;
    }

    /* Call the specific function to handle the request */
    if (ifData->im_if_data_type == IM_IF_GENERAL_STATS)
        *status = cwan_poseidon_set_gen_stats(hwidb,

&ifData->im_if_data.im_if_gen_stats);
    else if (ifData->im_if_data_type == IM_IF_PACKET_STATS)
        *status = IM_FAIL; /* not supported */
    else if (ifData->im_if_data_type == IM_IF_EXTENSION_STATS)
        *status = cwan_poseidon_set_ext_stats(hwidb,

&ifData->im_if_data.im_if_ext_stats);
    else if (ifData->im_if_data_type == IM_IF_UTILIZATION_STATS)
        *status = IM_FAIL; /* not supported */
    else {
        if (cwan_ifmgr_error_debug) {
            buginf("Warning!: Invalid if_data_type\n");
        }
    }
    return TRUE;
}
```

29.2.1.3.7 Interface Receive Address Data

The interface driver is responsible for implementing the `im_driver_if_rcvaddr_enumerate_callback()` service to make available specific receive address interface data to the CIM. It is important to note that `im_driver_if_rcvaddr_enumerate_callback()` is a CASE_LOOP service, which means that one or more driver can be registered with the same signature; therefore, each driver is responsible for making sure that the driver ID being passed correspond to them. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface (for example, HWIDB, SWIDB and subblock).

The status field should be used by the interface driver to indicate any error while retrieving the specific data.

Get Receive Address Data Example

This example shows a generic function to get a receive address data from a specific interface:

```
boolean
xx_if_get_rcvaddr_data (im_driver_id_t id, im_if_handle_t ifHandle,
 ulong rcvAddrIndex, unsigned char *rcvAddrBuff, ulong rcvAddrBuffMaxLength,
 ulong *rcvAddrBuffLength, im_if_rcvaddr_type_t * rcvAddrType,
 im_status_t *status)
{
    hwidbtype *hwidb;
    *status = IM_FAIL;

    /*
     * Check if this driveID corresponds to this driver. The driverID was
     * previously obtained during the subsystem init and should be global.
     */
    if (driverID != id) {
        return FALSE;
    }

    hwidb = (hwidbtype*)ifHandle;
    if (!hwidb) {
        return TRUE;
    }

    if ((rcvAddrBuff == NULL) || (rcvAddrBuffLength == NULL) ||
        (rcvAddrType == NULL || status == NULL))
    {
        return TRUE;
    }

    /* Update the rcvAddr params for CIM */
    *rcvAddrBuffLength = IEEEBYTES;
    memcpy(rcvAddrBuff, get_hwmacaddr(MACADDR_HARDWARE, hwidb),
           *rcvAddrBuffLength);
    rcvAddrType = IM_VOLATILE;

    return TRUE;
}
```

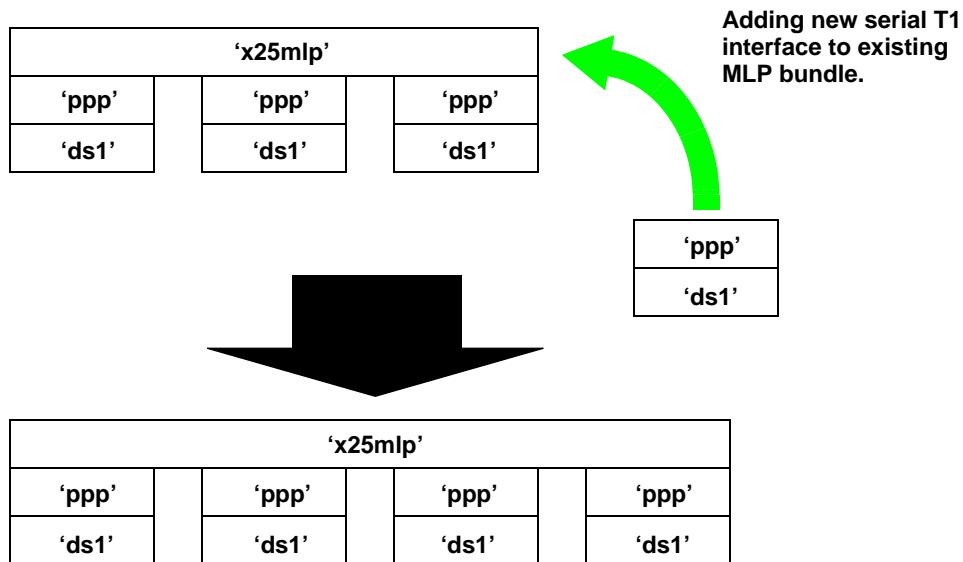
29.2.1.3.8 Interface Stack Relationships

Under normal conditions, the CIM automatically creates these stack relationships as long as the driver provides the interface parent index while creating an interface via the `im_driver_if_create()` service (see section “29.2.1.3.2 Interface Creation”). This section will show you how to manually create the stack for those special cases in which the CIM cannot automatically create it.

Adding an Interface Stack Relationship

There will be occasions when an interface driver cannot build an interface stack from the bottom-up. For example, consider an interface driver supporting MLP. In this example, the driver can create new serial interfaces and add them to an MLP bundle, as illustrated in Figure 29-5.

Figure 29-5 Adding an Interface to an MLP Bundle



The `im_driver_if_stack_relationship_add()` service supports the notion of adding interfaces to an arbitrary position within an existing interface stack.

When invoking this service, the interface manager requires the following data from the interface driver:

- `SuperiorInterface` —specifies the ifIndex value associated with the superior interface in the relationship being added.
- `SubordinateInterface` —specifies the ifIndex value associated with the subordinate interface in the relationship being added.

The interface manager imposes a single constraint on the invocation of this service. A driver (or drivers) must continue to adhere to the rule that interface stacks be created from the bottom-up, after which a driver (or drivers) may explicitly add stack relationships. Consider the example illustrated in Figure 29-5. In this example, the following sequence would be valid:

- Some driver invokes the `im_driver_if_create()` service to create the DS-1 interface.
- Some driver invokes the `im_driver_if_create()` service to create the PPP session over the DS1 interface.

- Some driver invokes the `im_driver_if_create()` service to create the MLP bundle. Observe that from the perspective of the interface manager, this represents a new interface stack.
- Some driver invokes the `im_driver_if_stack_relationship_add()` service to create a connection between the MLP bundle and the previously created PPP session.

Deleting an Interface Stack Relationship

Likewise, there will be occasions when an interface cannot tear down an interface stack from the top-down. The `im_driver_if_stack_relationship_delete()` service supports the notion of deleting interfaces from an arbitrary position within an interface stack. When invoking this use case, the interface manager requires the following data from the interface driver:

- Superior Interface —specifies the ifIndex value associated with the superior interface in the relationship being deleted.
- Subordinate Interface —specifies the ifIndex value associated with the subordinate interface in the relationship being deleted.

The interface manager imposes a single constraint on the invocation of this service. A driver (or drivers) must delete the stack relationship that it explicitly added before tearing down an interface stack (which it still must do from the top-down). Consider the example illustrated in Figure 29-5. In this example, the following sequence would be valid:

- Some driver invokes the `im_driver_if_stack_relationship_delete()` service to destroy the previously established connection between the MLP bundle and a previously created PPP session.
- Some driver invokes the `im_driver_if_destroy()` service to destroy the PPP session.
- Some driver invokes the `im_driver_if_destroy()` service to destroy the DS1 interface.

29.2.1.3.9 Interface Event Notification

An interface driver submits an event for processing by the interface manager by invoking the `im_driver_event_generate()` service. When invoking this service, the interface manager requires the following data from the interface driver:

- Interface —the ifIndex value associated with the interface to which the event relates.
- Event Type —an opaque value identifying the type of event that has occurred. This value only has interface-specific meaning (that is, the interface-specific extension and any event clients that subscribe to the particular type of event).
- Event Data —an opaque block of data indicating data relevant to the event. This value only has interface-specific meaning (that is, the interface-specific extension and any event clients that subscribe to the particular type of event).

The `im_driver_event_generate()` service prompts the interface manager to create an instance of an event object which it promptly places on the event queue. The interface manager introduces the event queue for two important reasons:

- There may be situations that result in a burst of event generation. The queue reduces the chances of a generated event being lost during these situations.
- An originator can generate an event with reduced latency, because the queue decouples the processing of the event from the notification.

Example (LinkUp Event)

This example shows part of the code that would build the linkup event message and notify the CIM of the event:

```
/* buffer size = sum of max sizes of each object +
 * (number of objects * 2)<-for type and length
 */
static char link_event_buffer[sizeof(im_if_admin_t) +
                               sizeof(im_if_oper_t) +
                               sizeof(ulong) + /* ifType */
                               (3 * 2)]; /* num objects * (type & len) */

static void
linkup_event_gen (ulong ifIndex)
{
    bytestream           event_data;
    uint                 length = 0;

    /* make TLV for ifAdmin */
    link_event_buffer[length++] = IM_IF_ADMIN_STATUS; /* type */
    link_event_buffer[length++] = 4;                   /* length */
    memcpy(&link_event_buffer[length], &ifAdmin, 4); /* value */
    length += 4;

    /* make TLV for ifOper */
    link_event_buffer[length++] = IM_IF_OPER_STATUS; /* type */
    link_event_buffer[length++] = 4;                   /* length */
    memcpy(&link_event_buffer[length], &ifOper, 4); /* value */
    length += 4;

    /* make TLV for ifType */
    link_event_buffer[length++] = IM_IF_TYPE;          /* type */
    link_event_buffer[length++] = 4;                   /* length */
    memcpy(&link_event_buffer[length], &ifType, 4); /* value */
    length += 4;

    event_data->length = length;
    event_data->ptr = link_event_buffer;

    /* Call CIM Client API to send the trap */
    reg_invoke_im_driver_event_generate(IM_EVENT_LINK_UP, ifIndex,
                                         event_data);
}
```

29.2.1.4 Development Unit Testing

At a minimum, the following unit tests must be performed when a new driver is adapted to use the new Interface Manager infrastructure.

29.2.1.4.1 Driver Registration

Ensure that driver can register with CIM, and that CIM recognizes the driver's signature.

29.2.1.4.2 Driver Creation/Destruction

The driver must ensure that proper creation and destruction takes place with the CIM. During any of these test cases, you must ensure that the IFMIB database is creating and destroying the appropriate interfaces. This can be done by using the **sh if-mgr db interface** command. The following procedures that could be executed include, but are not limited to:

- 1 Insert line card.
- 2 Remove line card.
- 3 Re-insert line card.
- 4 Repeat 1-3 using the **hardware remove/reload** command.
- 5 If applicable, repeat 1-4 using subinterfaces.

29.2.1.4.3 ifTable/ifXTable

The driver must ensure that proper interface-specific data can be retrieved or modified via the ifTable and the ifXTable.

- 1 Perform a **get** on all the applicable objects for the specific interface driver.
- 2 Perform a **getnext** on all the applicable objects for the specific interface driver.
- 3 Perform a **set** on all the applicable objects for the specific interface driver.

29.2.1.4.4 ifStackTable

If applicable, proper sub-layer stacking should be reflected in the ifStackTable. For interfaces that don't have any sub-layer, the ifStackTable should show always two entries 0.x and x.0.

29.2.1.4.5 ifAddressTable

The driver must ensure that proper interface address data can be retrieved or modified via the ifAddressTable.

- 1 Perform a **get** on all the applicable objects for the specific interface driver.
- 2 Perform a **getnext** on all the applicable objects for the specific interface driver.

29.2.1.4.6 IF Event Notification

The driver must ensure that proper notification is sent out when an interface or sub-layer interface changes operational status.

- 1 Create a condition to change the status of an interface from up to down.
- 2 Create a condition to change the status of an interface from down to up.

If applicable, repeat 1-2 for sub-layer interfaces.

29.2.2 Ethernet Interface Manager

The Ethernet Interface Manager (EIM) in an infrastructure that provides an interface-specific extension to the CIM for interfaces of type `ethernetCsmacd`. The EIM provides a control point that allows any management client (for example, ETHERLIKE-MIB) that requires configuring,

updating, monitoring or diagnosing a specific ethernet interface to access the data in a consistent way. It also allows interface drivers to make available (to the clients) their Ethernet-specific data in a consistent way without having to expose their internal data structures.

The EIM infrastructure is available starting with release 12.2(33)SRB. The EIM infrastructure is designed to be used in conjunction with the CIM infrastructure.

For specifics of the CIM architecture and how interface-specific managers are built around services provided by the CIM, see section 29.2.1 “Interface Manager Infrastructure”.

EIM APIs and public definitions are available in the /vob/ios/sys/if_mngr/ethernet directory.

29.2.2.1 Ethernet Interface Driver

An Ethernet interface driver is any subsystem maintaining the notion of a physical Ethernet interface. An Ethernet driver interacts with the EIM through APIs to implement the `eim_driver_registry.reg` callbacks to get/set/validate specific management data maintained by the driver.

29.2.2.1.1 Ethernet Interface Driver Initialization

Upon initialization, the Ethernet interface drivers must register the callbacks that implement the get/set/validate APIs defined in `eim_driver_registry.reg`. Note that the EIM does not provide services for driver registration or interface creation/destructions. These services are already provided by the CIM.

Ethernet Interface Driver Initialization Example

The following example shows a registration of a driver that supports Ethernet interfaces. The APIs highlighted in boldface are those EIM-supported APIs.

```
static void
spa_enet_mib_init_registries (void) {
    im_status_t status;
    if (driverID) {
        return;
    }
    /* register spa driver with Common Interface Manager */
    reg_invoke_im_driver_register(IM_ETHERNET, &driverID, &status);
    if (status == IM_PASS) {
        (void) reg_add_im_driver_if_data_get_callback (IM_ETHERNET,
ethernet_spa_if_get_data,"ethernet_spa_if_get_data");
        (void) reg_add_im_driver_if_data_validate_callback (IM_ETHERNET,
ethernet_spa_if_validate_data,"ethernet_spa_if_validate_data");
        (void) reg_add_im_driver_if_data_set_callback (IM_ETHERNET,
ethernet_spa_if_set_data,"ethernet_spa_if_set_data");
        /* Register get/validate/set callbacks with EIM */
        (void) reg_add_eim_driver_data_get_callback(IM_ETHERNET,
ethernet_spa_if_get_dot3_data,"ethernet_spa_if_get_dot3_data");
        (void) reg_add_eim_driver_collision_data_get_callback(IM_ETHERNET,
ethernet_spa_if_get_collision_dot3_data,
"ethernet_spa_if_get_collision_dot3_data");
        (void) reg_add_eim_driver_data_validate_callback(IM_ETHERNET,
```

```

        ethernet_spa_if_valid_dot3_data,
        "ethernet_spa_if_valid_dot3_data");
    (void) reg_add_eim_driver_data_set_callback(IM_ETHERNET,
        ethernet_spa_if_set_dot3_data, "ethernet_spa_if_set_dot3_data");
} else {
    IM_ERR_ENETEXT_DEBUG( "\nWarning! : __FUNCTION__ Registration
failed\n" );
}
}
```

29.2.2.1.2 Ethernet Interface Get Data

The interface driver is responsible for implementing the `eim_driver_data_get_callback()` and `eim_driver_collision_data_get_callback()` services in order to make available specific ethernet interface data to the EIM. It is important to note that

`eim_driver_data_get_callback()` is a CASE_LOOP service, which means that there can be one or more drivers registered with the same signature; therefore, each driver is responsible for making sure that the driver ID being passed correspond to it. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface.

Using the `eim_if_mgmt_data_t` data structure, the Ethernet interface driver can determine which set of data the EIM is requesting, and which field(s) within the set of data it needs to retrieve. If one or more fields are not available or not supported by the interface driver, then the driver can clear the specific fields.

The status field should be used by the Ethernet interface driver to indicate any errors while retrieving the specific data.

Get Ethernet Data Example

```

static boolean
ethernet_spa_if_get_dot3_data (im_driver_id_t id, im_if_handle_t ifHandle,
                               eim_if_mgmt_data_t *eimData, im_status_t* status)
{
    hwidbtype *hwidb = NULL;

    if (driverID != id) {
        IM_ERR_ENETEXT_DEBUG( "Warning!: ID(%d) does not match this driver
ID(%d)\n", id, driverID);
        return FALSE;
    }

    *status = IM_FAIL;

    hwidb = (hwidbtype*)ifHandle;
    if (!hwidb) {
        IM_ERR_ENETEXT_DEBUG( "Waning!: Invalid handle for this driver\n");
        return TRUE;
    }
    switch (eimData->eim_if_data_type) {
        case EIM_DOT3_GEN_STATS:
            *status = ethernet_spa_eim_get_dot3_stats(hwidb,
                                                       &eimData->eim_if_data.eim_dot3_gen_stats);
            break;
        case EIM_DOT3_PAUSE_DATA:
            *status = ethernet_spa_eim_get_dot3_pause(hwidb,
                                                       &eimData->eim_if_data.eim_dot3_pause);
            break;
    }
}

```

```

                &eimData->eim_if_data.eim_dot3_pause_stats);
        break;
    case EIM_DOT3_CONTROL_DATA:
        *status = ethernet_spa_eim_get_dot3_control(hwidb,
                                                     &eimData->eim_if_data.eim_dot3_control_stats);
        break;
    case EIM_DOT3_HC_GEN_STATS:
        *status = ethernet_spa_eim_get_dot3_hc_stats(hwidb,
                                                       &eimData->eim_if_data.eim_dot3_gen_stats);
        break;
    default:
        IM_ERR_ENETEXT_DEBUG("Warning!: Invalid eif_data_type
                             (%d)\n", eimData->eim_if_data_type);
        return (TRUE);
    }

    return (TRUE);
}

static im_status_t
ethernet_spa_eim_get_dot3_stats (hwidbtype *hwidb, eim_dot3_stats_t *stats_p)
{
    /* Clear the data values and rsp bit fields */
    memset (&(stats_p->eim_dot3_stats_values), 0,
            sizeof (eim_dot3_stats_values_t));
    memset (&(stats_p->eim_dot3_stats_rsp), 0,
            sizeof(eim_dot3_stats_fields_t));

    /* Call into jacket card to get the stats from the SPA */
    reg_invoke_spa_ether_mib_get_dot3_data_callback (hwidb,
EIM_DOT3_GEN_STATS, (void *) &(stats_p->eim_dot3_stats_values), (void *)
&(stats_p->eim_dot3_stats_rsp));

    /* dot3StatsIndex */
    if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsIndex)) {
        EIM_DOT3STATS_VALUE(stats_p, dot3StatsIndex) = hwidb->snmp_if_index;
        EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, dot3StatsIndex);
    }

    /* dot3StatsAlignmentErrors */
    if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsAlignmentErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsAlignmentErrors) {
        EIM_DOT3STATS_VALUE(stats_p, dot3StatsAlignmentErrors) =

stats_p->eim_dot3_stats_values.dot3StatsAlignmentErrors;
        EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, dot3StatsAlignmentErrors);
    }
    /* dot3StatsFCSErrors */
    if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsFCSErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsFCSErrors) {
        EIM_DOT3STATS_VALUE(stats_p, dot3StatsFCSErrors) =
            stats_p->eim_dot3_stats_values.dot3StatsFCSErrors;
        EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, dot3StatsFCSErrors);
    }

    /* dot3StatsSingleCollisionFrames */
    if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsSingleCollisionFrames)
&& stats_p->eim_dot3_stats_rsp.dot3StatsSingleCollisionFrames) {
        EIM_DOT3STATS_VALUE(stats_p, dot3StatsSingleCollisionFrames) =

```

```

stats_p->eim_dot3_stats_values.dot3StatsSingleCollisionFrames;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsSingleCollisionFrames);
}

/* dot3StatsMultipleCollisionFrames */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p,
dot3StatsMultipleCollisionFrames) &&
stats_p->eim_dot3_stats_rsp.dot3StatsMultipleCollisionFrames) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsMultipleCollisionFrames) =

stats_p->eim_dot3_stats_values.dot3StatsMultipleCollisionFrames;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsMultipleCollisionFrames);
}

/* dot3StatsSQETestErrors */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsSQETestErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsSQETestErrors) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsSQETestErrors) =
        stats_p->eim_dot3_stats_values.dot3StatsSQETestErrors;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsSQETestErrors);
}

/* dot3StatsDeferredTransmissions */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsDeferredTransmissions) &&
stats_p->eim_dot3_stats_rsp.dot3StatsDeferredTransmissions) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsDeferredTransmissions) =

stats_p->eim_dot3_stats_values.dot3StatsDeferredTransmissions;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsDeferredTransmissions);
}

/* dot3StatsLateCollisions */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsLateCollisions) &&
stats_p->eim_dot3_stats_rsp.dot3StatsLateCollisions) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsLateCollisions) =
        stats_p->eim_dot3_stats_values.dot3StatsLateCollisions;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsLateCollisions);
}

/* dot3StatsExcessiveCollisions */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsExcessiveCollisions) &&
stats_p->eim_dot3_stats_rsp.dot3StatsExcessiveCollisions) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsExcessiveCollisions) =
        stats_p->eim_dot3_stats_values.dot3StatsExcessiveCollisions;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsExcessiveCollisions);
}

/* dot3StatsInternalMacTransmitErrors */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p,
dot3StatsInternalMacTransmitErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsInternalMacTransmitErrors) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsInternalMacTransmitErrors) =

```

```
stats_p->eim_dot3_stats_values.dot3StatsInternalMacTransmitErrors;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsInternalMacTransmitErrors);
}

/* dot3StatsCarrierSenseErrors */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsCarrierSenseErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsCarrierSenseErrors) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsCarrierSenseErrors) =
        stats_p->eim_dot3_stats_values.dot3StatsCarrierSenseErrors;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsCarrierSenseErrors);
}

/* dot3StatsFrameTooLongs */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsFrameTooLongs) &&
stats_p->eim_dot3_stats_rsp.dot3StatsFrameTooLongs) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsFrameTooLongs) =
        stats_p->eim_dot3_stats_values.dot3StatsFrameTooLongs;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsFrameTooLongs);
}

/* dot3StatsInternalMacReceiveErrors */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p,
dot3StatsInternalMacReceiveErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsInternalMacReceiveErrors) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsInternalMacReceiveErrors) =
        stats_p->eim_dot3_stats_values.dot3StatsInternalMacReceiveErrors;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsInternalMacReceiveErrors);
}

/* dot3StatsSymbolErrors */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsSymbolErrors) &&
stats_p->eim_dot3_stats_rsp.dot3StatsSymbolErrors) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsSymbolErrors) =
        stats_p->eim_dot3_stats_values.dot3StatsSymbolErrors;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsSymbolErrors);
}

/* dot3StatsDuplexStatus */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsDuplexStatus) &&
stats_p->eim_dot3_stats_rsp.dot3StatsDuplexStatus) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsDuplexStatus) =
        stats_p->eim_dot3_stats_values.dot3StatsDuplexStatus;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsDuplexStatus);
}

/* dot3StatsRateControlAbility */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsRateControlAbility) &&
stats_p->eim_dot3_stats_rsp.dot3StatsRateControlAbility) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsRateControlAbility) =
        stats_p->eim_dot3_stats_values.dot3StatsRateControlAbility;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsRateControlAbility);
```

```

    }

/* dot3StatsRateControlStatus */
if (EIM_GET_DOT3STATS_REQ_FIELD(stats_p, dot3StatsRateControlStatus) &&
stats_p->eim_dot3_stats_rsp.dot3StatsRateControlStatus) {
    EIM_DOT3STATS_VALUE(stats_p, dot3StatsRateControlStatus) =
        stats_p->eim_dot3_stats_values.dot3StatsRateControlStatus;
    EIM_SET_DOT3STATS_RSP_FIELD_VALID(stats_p, \
                                      dot3StatsRateControlStatus);
}
return (IM_PASS);
}

```

29.2.2.1.3 Ethernet Interface Validate Data

The interface driver is responsible for implementing the `eim_driver_data_validate_callback()` service to validate configuration specific interface data that it is about to be changed. It is important to note that `eim_driver_data_validate_callback()` is a CASE_LOOP service, which means that one or more driver can be registered with the same signature; therefore, each driver is responsible for making sure that the driver ID being passed corresponds to it. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface.

Using the `eim_if_mgmt_data_t` data structure, the interface driver can determine which set of data it needs to validate. If one or more fields are not allowed to be changed or not supported by the driver interface, then the driver can clear the specific fields that it can support for the specific set of data.

The status field should be used by the interface driver to indicate any error while retrieving the specific data.

Validate Ethernet Data Example

Note The following example is for a driver that does not support writing/setting of management data.

```

static boolean
ethernet_spa_if_valid_dot3_data (im_driver_id_t id, im_if_handle_t ifHandle,
                                 eim_if_mgmt_data_t *eimData, im_status_t* status)
{
    hwidbtype *hwidb = NULL;
    if (driverID != id) {
        return FALSE;
    }
    hwidb = (hwidbtype*)ifHandle;
    if (!hwidb) {
        *status = IM_FAIL;
        return TRUE;
    }
    /* We do not support writes of dot3PauseAdminMode */
    *status = IM_NOT_WRITABLE;
    return (TRUE);
}

```

29.2.2.1.4 Ethernet Interface Set Data

The interface driver is responsible for implementing the `eim_driver_data_set_callback()` service to set specific Ethernet interface data (for example, admin Status and trap enable). It is important to note that `eim_driver_if_data_set_callback()` is a CASE_LOOP service, which means that there can be one or more driver registered with the same signature; therefore, each driver is responsible for making sure that the driver ID being passed correspond to them. Once the driver verifies the driver ID, it must cast the handle to the appropriate data structure type that represents the construct of the interface (for example, HWIDB, SWIDB and subblock).

Using the `eim_if_mgmt_data_t` data structure, the interface driver can determine which field(s) within the set of data it needs to set. If one or more fields are not available or it can not be modified by the driver interface, then the driver can clear the specific fields that it can support for the specific set of data.

The status field should be used by the interface driver to indicate any error while setting the specific data.

Set Ethernet Data Example

Note The following example is for a driver that does not support writing/setting of management data.

```
ethernet_spa_if_set_dot3_data (im_driver_id_t id, im_if_handle_t ifHandle,
                               eim_if_mgmt_data_t *eimData, im_status_t* status)
{
    hwidbtype *hwidb = NULL;
    if (driverID != id) {
        return FALSE;
    }
    hwidb = (hwidbtype*)ifHandle;
    if (!hwidb) {
        IM_ERR_ENETEXT_DEBUG("Warning!: Invalid handle for this driver\n");
        *status = IM_FAIL;
        return TRUE;
    }
    /* We do not support writes */
    *status = IM_FAIL;
    return (TRUE);
}
```

29.2.3 IF-MIB Infrastructure

This infrastructure applies to releases prior to 12.2SRA and 12.2SB.

With the addition of the IF-MIB (RFC 2233), Cisco IOS software can support subinterfaces in the interfaces group of MIB-II as well as the main interfaces.

The primary data structure holding the `ifmib` information related to various interfaces is the `struct snmpidbtype` defined in the `h/snmp_interface.h` file. The main fields in this structure are as follows:

```
struct snmpidbtype_ {
    ...
    ...
}
```

```

    struct snmpidbtype_ *next; /*pointer to next snmpidb in free list*/
    long snmp_if_index; /* what is the SNMP i/f index */

    enum SNMP_IF_DATA_TYPE snmp_if_struct; /* what kind of interface is it?*/
    snmpifptrtype snmp_if_ptr; /* pointer to the interface */

    ...
    ...
};


```

The `snmp_if_struct` tells what kind of interface the `snmpidb` contains. For example, in the case of main interfaces, it contains `SNMP_IF_HWIDB`.

`snmp_if_ptr` points to the actual interface-specific data. Again, taking the example of main interfaces, this points to the `hwidb` of the main interface.

Currently, IF-MIB `snmpidbs` are maintained as Red-Black Tree, indexed with `snmp_if_index`.

The primary data structure used for sublayers is the `subiabtype`, which is also defined in `h/snmp_interface.h`.

29.2.3.1 Supporting Main Interfaces, Controllers, and Subinterfaces

Supporting main interfaces, controllers, and subinterfaces is the same in IF-MIB with regard to the tables and API. This is explained in “29.2.3.1.1 IF_MIB Tables” and “29.2.3.1.2 IF-MIB API”. But the service points explained in “29.2.3.2.6 Adding to Service Points for Sublayers” are relevant only for the sublayers in IF-MIB.

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. This section describes these components and tells you how to use them in registering or deregistering sublayers. Then a sample implementation is provided, which uses Frame Relay sublayers. Furthermore, there is some information about link up/down trap support.

29.2.3.1.1 IF_MIB 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.

29.2.3.1.2 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.

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.

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.

h/snmp_interface.h

This file contains the `snmpidbtype` and the `subiabtype` data structures. Be sure that you study and understand these structures. These are the structures that you use to pass information across the IF-MIB API.

ifType

The `ifType` for a given subinterface is now supported in the `IANAifType` Textual Convention (TC). See `MIBS/IANAifType-TC.my`. 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 without planning to work on the IF_MIB requirements because 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 website. The `IANAifType-MIB` should be checked at the IANA website at <http://www.iana.org/assignments/ianaiftype-mib>.

29.2.3.2 Adding Support to Register or Deregister with IF-MIB

This section describes how to register and deregister main interfaces, controllers, and sublayers with IF-MIB.

29.2.3.2.1 Registering Main Interfaces

Main interfaces are registered with IF-MIB using the `reg_invoke_register_hwidb(hwidb)` registry call. `hwidb` is the hardware IDB of the interface to be registered. This API creates the `snmpidb`, sets the `snmp_if_struct` to `SNMP_IF_HWIDB`, and the `snmp_if_ptr` to the `hwidb` (argument to the registry call). After this, it inserts the newly created `snmpidb` into the IF-MIB RB tree.

29.2.3.2.2 Deregistering Main Interfaces

Main interfaces are deregistered with the `reg_invoke_deregister_hwidb(hwidb)` service point. This removes the `snmpidb` from the RB tree. The fields in `snmpidb` are zeroed out and the `snmpidb` is enqueued to the `snmpfreeidb` queue. This is mainly to avoid frequent mallocs and frees. Any new request for creating a new `snmpidb` is served from this queue if it is not empty. Otherwise, a new `snmpidb` is malloc'ed.

29.2.3.2.3 Registering Controllers

Controllers are registered with IF-MIB using the following service point:

```
boolean reg_invoke_ifmib_register_subif (
    snmpifptrtype *subif_ptr,
    enum SNMP_IF_DATA_TYPE subif_type,
    enum SNMP_ADMIN_STATE initAdminStatus)
```

Controllers are registered with `SNMP_IF_SUBIAB` as `subif_type`. Hence, the caller has to create a `subiab` structure of type `subiabtype` (see the `h/snmp_interface.h` file):

```
struct subiabtype_ {
    void *data_ptr; /* data ptr, used by vectored functions */

    snmpifptrtype master_ptr; /* used to point back at
                                controlling idb (hwidb or
                                swidb) -only for sublayers that
                                have no associated idbs (ie
                                aal5 layer) */

    enum SNMP_IF_DATA_TYPE master_type; /* what kind of interface is
                                         master_ptr */

    ...
    ...

}
```

For controllers, `master_type` must be set as `SNMP_IF_CDB` and `master_ptr` must be initialized to point to the `cdb` of the controller being registered.

After creating the `subiab` for controller registration, the `subiab` has to be embedded in `snmpifptrtype` (see the `h/snmp_interface.h` file) and the above registry should be invoked with `subif_type` as `SNMP_IF_SUBIAB`.

29.2.3.2.4 Deregistering Controllers

Use the `reg_invoke_ifmib_deregister_subif (long ifIndex)` service point for deregistering controllers.

29.2.3.2.5 Registering and Deregistering Sublayers

The primary data structure used for sublayers is the `subiabtype`, as defined in the `h/snmp_interface.h` file. You should use this structure for any interfaces or sublayers that are not `hwidb`-based. The next sections describe how to correctly support your new sublayer in the IF-MIB, as well as registering and deregistering sublayers with IF-MIB.

29.2.3.2.6 Adding to Service Points for Sublayers

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 `RETRVAL` or `LIST` type service points use the `ifType` as the selection criterion. 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 `edb->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 `edb->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. A new function must be added either if the new sublayer needs to support the `ifAdminStatus` in a sublayer specific fashion, or even if the sublayer doesn't support modifications to `ifAdminStatus`. In the former case, the function must add code to perform the admin status change, while in the latter case the function returns `FALSE` without any processing.

- `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_ifmib_create_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_ifmib_create_stacklink()` or `reg_invoke_ifmib_destroy_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()` to register a sublayer. Use `ifmib_deregister_subif()` to de-register a sublayer.

- `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_ifmib_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.

29.2.3.2.7 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 `ifPhysAddr`, but it is up to you, the developer, to determine when the address is known and to make a call to `reg_invoke_ifmib_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`.

29.2.3.2.8 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 freed.

29.2.3.2.9 Modifying the ifRcvAddressTable for Sublayers

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_ifmib_create_rcvaddr()` or `reg_invoke_ifmib_destroy_rcvaddr()`. These functions simply add or remove entries into the `ifRcvAddressTable` with little sanity checking.

29.2.3.2.10 Modifying the ifStackTable for Sublayers

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_ifmib_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_ifmib_create_stacklink()` or `reg_invoke_ifmib_destroy_stacklink()`. These functions simply add or remove entries into the `ifStackTable` with little sanity checking.

29.2.3.2.11 Sparse Table Support for Sublayers

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.

29.2.3.3 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.

29.2.3.3.1 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.

29.2.3.3.2 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.

29.2.3.3.3 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.

29.2.3.4 Link Up/Down Notification 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.

29.3 Entity MIB Infrastructure

The following information was taken from developers, and the Entity MIB version 2 - ENG-77737, and CISCO-ENTITY-EXT-MIB Functional/Design Specification - ENG 106791 documentation.

This section describes the Entity MIB version 2 (RFC 2737), which replaces the existing Entity MIB version 1(RFC 2037). In summary, a need for a standardized way of representing a single SNMP agent which supports multiple instances of one MIB was requested. The following ENTITY-MIBs address this issue.

29.3.1 Entity MIB API

Use the following functions to provide an entity MIB infrastructure.

29.3.1.1 Functions to Add Entity Objects to the ENTITY-MIB

29.3.1.1.1 Adding a Physical Entity to the ENTITY-MIB

The [reg_invoke_entityapi_add_physical_entity_struct\(\)](#) function adds a physical entity to the ENTITY-MIB.

```
SR_UINT32 reg_invoke_entityapi_add_physical_entity_struct (const entPhyType  
*phyEntry);
```

This function replaces [reg_invoke_entityapi_delete_physical_entity\(\)](#).

29.3.1.1.2 Adding a Logical Entity to the ENTITY-MIB

The [reg_invoke_entityapi_add_logical_entity_struct\(\)](#) function adds a logical entity to the ENTITY-MIB.

```
SR_UINT32 reg_invoke_entityapi_add_logical_entity_struct (const  
entLogicalType *logEntry);
```

This function replaces [reg_invoke_entityapi_add_logical_entity\(\)](#).

29.3.1.1.3 Adding Logical Entity - Physical Entity IP Mapping

The [reg_invoke_entityapi_add_lpmap\(\)](#) function adds logical entity-physical entity mapping.

```
void reg_invoke_entityapi_add_lpmap (SR_UINT32 logicalIndex, SR_UINT32  
physicalIndex);
```

29.3.1.1.4 Adding Logical and Physical Entity Mapping Alias

The [reg_invoke_entityapi_add_alias\(\)](#) function adds the logical and physical entity mapping alias.

```
void reg_invoke_entityapi_add_alias (SR_UINT32 physicalIndex, SR_UINT32  
logicalIndex, OID *aliasMappingIdentifier);
```

29.3.1.2 Functions to Delete Entity Objects from the ENTITY-MIB

29.3.1.2.1 Deleting a Physical Entity from the ENTITY-MIB

The [reg_invoke_entityapi_delete_physical_entity\(\)](#) function deletes a physical entity from the ENTITY-MIB.

```
void reg_invoke_entityapi_delete_physical_entity (SR_UINT32 physicalIndex);
```

29.3.1.2.2 Deleting a Logical Entity from the ENTITY-MIB

The [reg_invoke_entityapi_delete_logical_entity\(\)](#) function deletes a logical entity from the ENTITY-MIB.

```
void reg_invoke_entityapi_delete_logical_entity (SR_UINT32 logicalIndex);
```

29.3.1.2.3 Deleting a Physical and Logical Entity Mapping from the ENTITY-MIB

The [reg_invoke_entityapi_delete_lpmap\(\)](#) function deletes a physical and logical entity mapping from the ENTITY-MIB.

```
void reg_invoke_entityapi_delete_lpmap (SR_UINT32 physicalIndex, SR_UINT32 logicalIndex);
```

29.3.1.2.4 Deleting a Logical and Physical Entity Mapping Alias

The [reg_invoke_entityapi_delete_alias\(\)](#) function deletes a logical and physical entity mapping alias.

```
void reg_invoke_entityapi_delete_alias (SR_UINT32 physicalIndex, SR_UINT32 logicalIndex);
```

29.3.1.3 Functions to Look Up Entity Objects in the ENTITY-MIB

29.3.1.3.1 Looking Up Physical Entity in the ENTITY-MIB

The [reg_invoke_entityapi_lookup_physical_entity\(\)](#) function gets the physical entity for a given physical index.

```
entPhyNode *reg_invoke_entityapi_lookup_physical_entity (SR_UINT32 physicalIndex, int searchType);
```

29.3.1.3.2 Looking Up Logical Entity in the ENTITY-MIB

The [reg_invoke_entityapi_lookup_logical_entity\(\)](#) function gets the logical entity for a given logical index.

```
entLogicalType *reg_invoke_entityapi_lookup_logical_entity (SR_UINT32 logicalIndex, int searchType);
```

29.3.1.3.3 Looking Up the IP Mapping in the ENTITY-MIB

The [reg_invoke_entityapi_lookup_lpmap\(\)](#) function gets the logical to physical entity mapping entry from the logical and physical indices given.

```
lpMappingType *reg_invoke_entityapi_lookup_lpmap (SR_UINT32 logicalIndex, SR_UINT32 physicalIndex, int searchType);
```

29.3.1.3.4 Looking Up the Alias in the ENTITY-MIB

The [reg_invoke_entityapi_lookup_alias\(\)](#) function gets the alias entity for a given logical and physical index combination.

```
aliasType *reg_invoke_entityapi_lookup_alias (SR_UINT32 physicalIndex, SR_UINT32 logicalIndex, int searchType);
```

29.3.1.3.5 Testing the Physical Entity in the ENTITY-MIB

The [reg_invoke_entityapi_test_physical_entity\(\)](#) function does a get-next search of physical entities starting with the physical index passed as an argument to the function and returns the next physical entity (if any) that evaluates to TRUE.

```
entPhyNode *reg_invoke_entityapi_test_physical_entity (SR_UINT32
physicalIndex,PhysicalEntityProcTest test,void *index2,int searchType);
```

29.3.1.4 Utility Functions

29.3.1.4.1 Finding the Vendor OID for a Particular HWIDB Interface

The [reg_invoke_entityapi_idbtype_to_vendoroid\(\)](#) function finds the vendor OID for a particular hwidb interface, if possible.

```
void reg_invoke_entityapi_idbtype_to_vendoroid (SR_UINT32 idbtype,OID
*vendor_oid,hwidbtype *hwidb);
```

29.3.1.4.2 Formatting IP Transport Information into a String

The [reg_invoke_entityapi_IpTransportInfoToString\(\)](#) function takes an IP address type pointer and a UDP port number and formats them into a string.

```
void reg_invoke_entityapi_IpTransportInfoToString (char *dst,ipaddrtype
address,short udpport);
```

29.3.1.5 Extension ENTITY-MIB Support

29.3.1.5.1 Adding a Physical Entity Extension Entry

The [reg_invoke_entityapi_add_physical_entity_extension\(\)](#) function adds a physical entity extension entry.

```
boolean reg_invoke_entityapi_add_physical_entity_extension (SR_UINT32
entPhyIndex,const ceExtProcessorEntry *proc, const ceExtConfigRegEntry
*conf);
```

29.3.1.5.2 Deleting a Physical Entity Extension Entry

The [reg_invoke_entityapi_delete_physical_entity_extension\(\)](#) function deletes a physical entity extension entry.

```
boolean reg_invoke_entityapi_delete_physical_entity_extension (SR_UINT32
entPhyIndex);
```

29.3.1.5.3 Looking Up a Physical Extension Entry

The [reg_add_entityapi_lookup_physical_entity_ext_processor\(\)](#) function looks up a physical extension entry which has the extension processor entry populated.

```
entPhyExtType *reg_add_entityapi_lookup_physical_entity_ext_processor
(SR_UINT32 physicalIndex,int searchType);
```

The `reg_add_entityapi_lookup_physical_entity_ext_confreg()` function looks up a physical extension entry which has the extension configuration registry entry populated.

```
entPhyExtType *reg_add_entityapi_lookup_physical_entity_ext_confreg(SR_UINT32
physicalIndex,int searchType);
```

29.4 MIB Persistence Infrastructure

The following information was taken from the developers and ENG-77738.

The MIB Persistence feature enables the SNMP data of a MIB to be persistent across reloads. The MIB desired to be persisted needs to be enabled for persistence by the following configuration command:

[no] snmp mib persist [mib-name]

The MIB data is actually written to non-volatile storage when the EXEC command **write mib-data** is invoked. Only a snapshot of the data is persisted when **write mib-data** is executed. The MIB data modified after that is not persisted unless the EXEC command is invoked again. At the time of reload, if persistence is enabled for a MIB, the MIB data is loaded from the associated file and the MIB is populated. The loading of the MIB does not affect the boot time because loading is done when the SNMP process comes up for the first time. When persisting the MIB data, the version information of the MIB is also stored so that at the time of reload if the data persisted is not the same as the MIB version supported by this image, the data is safely ignored.

29.4.1 MIB Persistence API

MIBs use the registry services provided by the persistence infrastructure for loading and storing the data to non-volatile storage. The persistence infrastructure consists of a process (`mib_persist_process`) that has the following registry functions `reg_added`.

29.4.1.1 Writing Data to NVRAM

The `reg_invoke_store_persistence_data()` function writes data to the persistent storage (NVRAM). The MIB uses this registry call at appropriate times to persist the state of the MIB.

```
void store_persistence_data(int persist_id, char* data, int length)
```

29.4.1.2 Loading Persistent Data

The `reg_invoke_load_persistence_data()` function reads data from the persistent storage (NVRAM). MIBs use this registry call during initialization of the MIB to load the persisted data.

```
mibData* load_persistence_data(char *mib_name)
```

29.4.1.3 Opening Underlying File in Non-volatile Storage Area

The `reg_invoke_open_persistence_storage()` function opens the underlying file in the non-volatile storage area.

```
int open_persistence_storage (char *mib_name, char *version, int version_len)
```

29.4.1.4 Closing Persistence Storage

Once writing of the MIB data is over, the MIBs should close the persistence storage. Not doing this may result in the making the underlying filesystem inaccessible on some platforms. The [reg_invoke_close_persistence_storage\(\)](#) function closes persistence storage.

```
void close_persistence_storage (int persist_id)
```

29.4.1.5 Deleting the Underlying File

When the persistence is disabled, and the **write mem** is done, the underlying file needs to be deleted. The [reg_invoke_delete_persistence_storage\(\)](#) function deletes persistence storage.

```
void delete_persistence_storage (char *mib_name)
```

29.4.1.6 Persistence Support for MIBs

The [reg_invoke_enable_mib_persistence\(\)](#) function is called by the CLI. It enables/disables the persistence by setting/resetting the global MIB specific variable.

```
void enable_mib_persistence (boolean sense, int flags)
```

The [reg_invoke_load_mib\(\)](#) function loads the persisted MIB data from NVRAM if persistence has been enabled for that MIB. If the persistence is not enabled for that particular MIB and if the associated file is in the underlying storage area, the file is removed.

```
void load_mib (void)
```

29.5 SNMP Notification Infrastructure

This section includes information on the SNMP Notification API and SNMP Notification Implementation.

29.5.1 SNMP Notification API

The following functions are available for adding a new `ifType` for a hardware interface (`hwidb`):

- 1 `isGeneralInfoGroupOnly()`
- 2 `ifType_get()`
- 3 `reg_add_hc_hwcounter_get()`
- 4 `reg_add_ifmib_get_operstatus_hwidb()`
- 5 `reg_add_ifmib_hwifSpecific_get()`

These functions are described in the next subsections.

Note In the following subsections, "**OPTIONAL : 122T**" means two things. "OPTIONAL" means that the modification of the function/addition of a registry is optional while "122T" signifies that this function/registry is supported (or invoked) only on 122T. Therefore, 120S provides no support for this function/registry.

29.5.1.1 `isGeneralInfoGroupOnly()`

OPTIONAL : 120S and 122T

This function needs to be modified if your interface supports only the objects in the `ifGeneralInformationGroup`. By default, the code assumes that all objects are supported—so if you only support the `ifGeneralInformationGroup` subset—you'll need to add code into this function.

29.5.1.2 `ifType_get()`
MANDATORY : 120S and 122T

You will need to add code to this function to return proper values when the query pertains to your interface. This is the function used to return the `ifType` value in the IF-MIB.

29.5.1.3 `reg_add_hc_hwcounter_get()`
OPTIONAL : 120S and 122T

This registry is invoked to get 64-bit values on platforms/interfaces that provide native 64-bit support. GSR might be the only platform which uses this registry. A default `reg_add` is done for this by the `hc_counter.c` code.

29.5.1.4 `reg_add_ifmib_get_operstatus_hwidb()`
OPTIONAL : 122T

This registry is invoked to get the operational status of the interface in question. A default, the registry is already added in the `sr_ifmib.c` file. If you need different functionality than what is provided, you'll need to register a callback. At the time of this writing, all interfaces seemed to use the default.

29.5.1.5 `reg_add_ifmib_hwifSpecific_get()`
OPTIONAL : 122T

This registry is used to populate the value of `ifSpecific`. By default, a null OID, meaning { 0 0 }, is returned.

29.5.2 SNMP Notification Implementation

There are two types of notifications:

- One type does not assure that the host has received the notification. This is known as Traps.
- Another type is *assured* notification because it sends the notification until the host receives it or else until the number of configured retries has been exhausted.

The implementation of SNMP Notification adheres to the standards defined in the `SNMP-NOTIFICATION-MIB` and `SNMP-TARGET-MIB`. Based on these standards, the following four tables are implemented to store the information required for the notifications; `snmpNotifyTable`, `snmpTargetAddrTable`, `snmpTargetParamsTable`, and `snmpNotifyFilterProfileTable`. These tables are described here:

- `snmpNotifyTable`: stores a unique name for each entry (`snmpNotifyName`), a single tag value for each entry that is used to select the entries in the `snmpTargetAddrTable`, the type of the notifications (informs or traps), the storage type, and the status.
- `snmpTargetAddrTable`: stores a unique identifier associated with each entry (`snmpTargetAddrName`), information about transport type of the address contained in the `snmpTargetAddrTAddress` (`snmpTargetAddrTDomain`), i.e. `snmpDomains`.

`snmpTargetAddrTAddress` stores the target IP address in the first four bytes and the `udp` port number in the next two bytes, stores the timeout value required to wait for the response in case of informs in `snmpTargetAddrTimeout`, then stores the retry count in `snmpTargetAddrRetryCount`.

`snmpTargetAddrTagList` contains a tag value that is used to select target addresses from the table.

`snmpTargetAddrParams` identifies the entry in the `snmpTargetParamsTable`, stores the storage type, and stores the status of the entry.

- `snmpTargetParamsTable`: stores a unique identifier associated with each entry in `snmpTargetParamsName`.

`snmpTargetParamsMPModel` has the message processing model number.

`snmpTargetParamsSecurityModel` gives the version of the trap to be generated.

`snmpTargetParamsSecurityName` has the community name specified during the configuration of the host.

`SnmpTargetParamsSecurityLevel` gives the level of security to be used when generating SNMP messages.

Then, this table stores the storage type and the status of the entry.

- `snmpNotifyFilterProfileTable`: stores the name of the filter profile to be used when generating notifications in `snmpNotifyFilterProfileName`, the storage type, and the status of each entry.

For more details on these tables refer to RFC 2573.

29.5.2.1 Flow of the Trap Generation

The following steps describe the flow of the trap generation:

- 1 When a MIB process has to generate traps, it invokes either `SnmpSendTrapWithValues()` or `SnmpSendTrap_Generic()` based on following points:
 - (a) `SnmpSendTrap_Generic()` invoking MIB need not know the varbind values. This makes the calling of the trap a lot simpler and can be called during an interrupt.
 - (b) `SnmpSendTrapWithValues()` invoking MIB should know the values of the varbind before invoking. `SnmpTrapProcess` will take care of freeing the memory allocated to the varbinds after sending the trap.

The functions above simply enqueue the trap request to `SnmpTrapQueue`.

- 2 `SnmpTrapProcess()` dequeues the trap request from the queue and invokes `do_trap()` directly if the values of varbinds are given, or else invokes `snmp_trap()`, which in turn does a `i_GetVar()` to fill in the values of the required varbinds and calls `do_trap`.
- 3 `do_trap()` calls a wrapper function `SrGenerateNotificationWrapper()`. This wrapper function invokes `SrV2GenerateNotification()`.
- 4 `SrV2GenerateNotification()` checks whether traps have to be generated or informs. For informs, it invokes the `snmp_do_inform()` (generation of informs is covered later) registry. For traps loops, `snmpNotifyTable` and checks the entry for traps and ignores for informs. Once it gets the entry for the trap, it loops `snmpTargetAddrTable` until it finds `snmpTargetAddrTagList` that matches with the `snmpNotifyTag` of the selected entry. Then, it gets the value of `snmpTargetAddrTDomain` and `snmpTargetAddrTAddress`, and fills the transport information structure. It extracts data from the `snmpTargetParamsTable`, which is

pointed by `snmpTargetAddrParams` variable of `snmpTargetAddrTable`. Then, it checks `snmpTargetParamsMPModel` with the version of the trap to be generated, it also checks `snmpTargetParamsSecurityModel`, `snmpTargetParamsSecurityLevel`, and `snmpTargetParamsSecurityName` (community). It builds the PDU and invokes `SrGenerateNotification()` passing the pointer to the entry in `SrGenerateNotification`, `snmpTargetAddrTable`, `snmpTargetParamsTable`, and the `pdu`.

- 5 `SrGenerateNotification()` fills the PDU with the varbinds, then checks the corresponding entry in `snmpNotifyFilterProfileTable`, filters out the trap depending on the filter configured, creates an SNMP message, and invokes `Sr_send_trap()`.
- 6 `Sr_send_trap` builds the SNMP message with the PDU and the transport information passed, and then invokes `SendToTransport()`.
- 7 `SendToTransport()` invokes the registry for `ip_snmp_trap()`. This registry builds the packet from the SNMP message passed and enqueues this packet to `loghost->queue` for the given host address.

29.5.2.2 Flow of the Informs

The following steps describe the flow of the informs:

- 1 The `snmp_do_inform` registry is invoked.
- 2 `snmp_do_inform()` searches the host entry for the given destination, fills in the `snmp_proxy_retry_t` structure, and invokes `snmp_proxy_docmd()` to send the pending informs.
- 3 After the inform is sent, the request is enqueued to the pending queue.
- 4 `snmp_inform_process_queue()` processes, dequeues `snmp_informQ` (queue for inform responses). Checks the response, and updates the retry counters.
- 5 If the inform has reached the host, the pending request is deleted or else it resends the inform until the host receives or until the configured retries are exhausted.

29.6 HA MIB Sync Infrastructure

The HA MIB Sync Infrastructure was designed as part of the IOS High Availability initiative to improve the availability of networks made up of Cisco IOS-based routers. In this context, high availability translates as the ability of the router to continue forwarding traffic, continue operating as a routing protocol peer, and remain manageable under a set of circumstances that without HA support would cause an interruption in service.

From a network management (SNMP) perspective, the goal of the HA MIB Sync Infrastructure is to provide an uninterrupted management interface to the end user. To accomplish this goal, everything visible to the manager should be copied across (to the standby), or the manager should recognize that it needs to update its view of the agent, during or before a fail-over to the redundant processor. In the case that the manager needs to update its view, the behavior of the MIBs that will/will not be preserved (in part or in whole) across a switchover needs to be documented and available to customers. Thus, the following HA MIB Sync Infrastructure support was completed in Phase 1 development:

- SNMP MIB Sync Infrastructure

- Syncing of IF-MIB, ENTITY-MIB, SNMP-USM-MIB, SNMP-VACM-MIB, SNMP-COMMUNITY-MIB (in Phase 2), SNMP-FRAMEWORK-MIB, and CISCO-RF-MIB
- Syncing of ifIndex and entPhysicalIndex
- Syncing of sysUptime
- Various Notification Supports During Switchover

This HA MIB Sync Infrastructure support was developed considering the following:

- Uninterrupted SNMP View to NMS
- Communication to the NMS
- General Issues Regarding SSO Functionality

The following sections provide an overview of the issues that came up with respect to SNMP on a switchover and the solutions that were implemented.

29.6.1 Uninterrupted SNMP View to NMS

Providing an uninterrupted SNMP view to the network management station across a switchover is accomplished by preparing the following:

- Maintaining sysUpTime across Switchover
- sysObjectID
- SNMP Configuration Across Switchover
- IF-MIB and ENTITY-MIB

The following subsections describe how to provide an uninterrupted SNMP view to the network management station across a switchover.

29.6.1.1 Maintaining sysUpTime across Switchover

From an SSO perspective, the standby system's NM engine should be indistinguishable from the active system's NM engine, so an NMS should not get the impression of a new device coming up whenever a switchover occurs.

Another important requirement for maintaining sysUpTime comes from the fact that the ILMI (Integrated Local Management Interface) operation depends on it.

One of the requirements of the NSF/SSO projects is to maintain ATM connections across a switchover. This requires that ILMI stay up across a switchover (or else ATM sessions would be lost). And ILMI must see its representation of sysUpTime as continuing to arch forward in time, i.e. increasing in value, or else it will reset.

To satisfy these requirements, sysUpTime is maintained across a switchover.

The system clock is synced by `set_slave_time()`. Since passive timers (`sys_timestamp`) are used for calculation of sysUpTime and passive timers use the system clock (which is already being synced up), no syncing of the sysUpTime value is needed.

Synchronization of sysUpTime was changed to be equivalent to the synchronization of system time.

Note that sysUpTime is an INTEGER value ranging from 0 to 4294967295, and represented in hundredths of a second. This translates to the fact that sysUpTime can represent a monotonically increasing value for 497.1 days, after which it will wrap around, resulting in resetting all ATM connections unless ILMI is wraparound-aware.

So, ILMI was made to be wraparound-aware, meaning:

- ILMI was changed to understand (based on the last queried value) that a wraparound is possible, so if the last value of sysUpTime is just below the wraparound, it should expect that the next value could be lower (and will not reset the connection).
- The ILMI query includes the snmpBoots value. Based on the value of snmpBoots, ILMI can evaluate whether there has been a reset of the router or a wraparound of the sysUpTime.

29.6.1.2 sysObjectID

The sysObjectID was developed to remain the same across a switchover, in spite of the differing processor types on the active and the standby. This means a RSP2 processor type and a RSP4 processor type can be paired because the sysObjectID represents a platform (such as the 7500) and will remain the same across a switchover because it represents the platform and not the processor type.

29.6.1.3 SNMP Configuration Across Switchover

While most of the SNMP configuration is synchronized with the standby during a (running/startup) config synchronization, the SNMP configuration done via SNMP using SNMP PDUs (Protocol Data Units) does not show up on the running configuration and hence could be lost on a switchover if not synchronized. Therefore, the state information is synced up across the active and the standby for the following SNMP infrastructure MIBs:

- 1 SNMPv2-MIB
- 2 SNMP-FRAMEWORK-MIB
- 3 SNMP-USM-MIB
- 4 SNMP-VACM-MIB
- 5 SNMP-TARGET-MIB
- 6 SNMP-COMMUNITY-MIB (in Phase 2, not supported in 12.0ST)
- 7 SNMP-NOTIFICATION-MIB (in Phase 2, not supported in 12.0ST)
- 8 SNMP-NOTIFICATION-LOG-MIB (in Phase 2, not supported in 12.0ST)
- 9 SNMP-FRU-CONTROL-MIB (in Phase 2, not supported in 12.0ST)

Some of the tables from the SNMP-USM-MIB, SNMP-TARGET-MIB and SNMP-COMMUNITY-MIB are persisted. Usage of the persistence infrastructure is defined in Phase 2.

Note The SNMP-PROXY-MIB is not in the list above because it is used mainly on the 6400 platform and per the ENG-61816 documentation, “IOS HA Enhancement Architecture”, the 6400 is not a target platform.

Synchronization is completed across even the non-active rows because a centralized dynamic sync requires that the previous state of the associated row be present on the standby for the sync to succeed.

29.6.1.4 IF-MIB and ENTITY-MIB

While these MIBs don't form a part of the core SNMP MIBs, these infrastructure MIBs are extensively used by other MIBs and network management applications.

Persistence of `ifIndices` across system reboots is an important customer requirement, and therefore it follows that reflecting the same `ifIndices` across a switchover is expected.

`ENTITY-MIB` is populated whenever a device boots up. Therefore, changes in `entityIndices` are perceived as a reboot of the device. Further entity indices are also used as indices in other MIBs, and the network management station needs to re-synchronize all the related information if the entity indices change.

Therefore, support of active to standby synchronization of `IF-MIB` and `ENTITY-MIB` was implemented using work done for the Harley project (see ENG-31829).

`IfIndex` persistence redesign for RPR and RPR+ is planned for Phase 2 development in case there are any interfaces that will reset and re-register (with the `IF-MIB`) on a switchover.

29.6.2 Communication to the NMS

This section describes the changes that are communicated to the manager during a switchover.

The changes include:

- Counters/Statistics
- `failoverTime/redundantSysUpTime`
- Switchover Notification

29.6.2.1 Counters/Statistics

Statistics and counter values are not synced over from the active to the standby RP since the amount of data that would need to be synced is huge and also changes rapidly. (Linecards would be clearing the statistics/counters registers during SSO switchover processing and this would result in a problem from a network management perspective, since managers rely on the continuity of counters, meaning no unknown/un-notified counter reset occurs.)

The manager usually re-synchronizes either on noticing a change in value of a discontinuity object (indicating the last time that a discontinuity of data occurred) or on receipt of a notification indicating `warmStart` or `coldStart` of the management agent. Generation of a `warmStart/coldStart` notification is accompanied by resetting of `sysUpTime`, which is disallowed due to the requirements of NSF/SSO (described in the section on `sysUpTime`).

On a discontinuity object front, there can be MIBs which do not support a discontinuity object and yet support counters/statistics which may get reset on a switchover. There may be a need for these MIBs to have a discontinuity object indicating the switchover time.

Syncing of `IF-MIB` counters is planned for Phase 3 development.

29.6.2.2 CISCO-RF-MIB

The `CISCO-RF-MIB` defines `ciscoRFSwactNotif` notification, which informs a manager about a switchover. This notification is used by the manager to resync the counter/statistics values.

To support the MIB discontinuity objects, `CISCO-RF-MIB` was modified to include a new discontinuity object, `failoverTime/redundantSysUpTime`, which indicates the value of `sysUpTime` when the switchover occurred. This object is used as a discontinuity marker for all MIBs that do not support a discontinuity object (and need one on switchover).

The need and usage of the `ciscoRFSwactNotif` from a counter perspective is as follows, per the `CISCO-RF-MIB` documentation:

DESCRIPTION

"A SWACT notification is sent by the newly active redundant unit whenever a switch of activity occurs. In the case where a SWACT event may be indistinguishable from a reset event, a network management station should use this notification to differentiate the activity.

sysUpTime is the same sysUpTime defined in the RFC-1213 MIB."

```
::= { ciscoRFMIBNotifications 1 }
```

The failoverTime/redundantSysUpTime indicates that every MIB counter that does not have a discontinuity object associated with it has observed a discontinuity at the time indicated by failoverTime/redundantSysUpTime. If certain MIBs do support synchronization of statistics/counters, but do not support any discontinuity object, even they are marked as having discontinued (which is not exactly correct). But this leads to no side-effects, except for the fact that the these counter values would (unnecessarily) be re-synced by the manager.

The need and usage of the cRFStatusFailoverTime OBJECT-TYPE is as follows, per the CISCO-RF-MIB documentation:

SYNTAX TimeStamp

MAX-ACCESS read-only

STATUS current

DESCRIPTION

"The value of sysUpTime when the primary redundant unit took over as active. The value of this object will be 0 till the first switchover."

```
::= { cRFStatus 9 }
```

Some MIBs already have a discontinuity object for indicating when the counter/statistic information suffered the last discontinuity. These MIBs need to be notified about the discontinuity on a switchover. The approach adopted to notify these MIBs was to use a registry call, the SNMP subsystem would do a reg_invoke() on the switchover discontinuity; each of the interested MIBs registers (reg_adds) for the notification and carries out required tasks.

Here is the list of MIBs that need to be added using reg_adds:

```
APPN-MIB  
ATM-RMON-MIB  
CISCO-SRP-MIB  
IF-MIB  
IPROUTE-MIB  
MPLS-LDP-MIB  
MPLS-LSR-MIB
```

The following MIBs depend on sysUpTime and, while it does not look like these need any modifications, you need to check with the authors of these MIBs:

```
ATM-MIB  
CISCO-BUS-MIB  
CISCO-CALL-HISTORY-MIB  
CISCO-CIPCMPC-MIB  
CISCO-CONFIG-MAN-MIB  
CISCO-DIAL-CONTROL-MIB  
CISCO-FRAS-HOST-MIB  
CISCO-FTP-CLIENT-MIB  
CISCO-IP-ENCRYPTION-MIB  
CISCO-IPROUTE-MIB
```

CISCO-ISDN-MIB
CISCO-LECS-MIB
CISCO-REPEATER-MIB
CISCO-RTTMON-MIB
CISCO-SNA-LLC-MIB
CISCO-SYSLOG-MIB
DIAL-CONTROL-MIB
DOCS-IF-MIB
ENTITY-MIB
EVENT-MIB
EXPRESSION-MIB
OLD-CISCO-CHASSIS-MIB
PIM-MIB
RFC1315-MIB
RFC1382-MIB
RMON-MIB
SNA-NAU-MIB
SNA-SDLC-MIB
SNMP-REPEATER-MIB
SNMPv2-MIB

29.6.2.3 failoverTime/redundantSysUpTime

To summarize, the `failoverTime/redundantSysUpTime` is defined in the `CISCO-RF-MIB` to:

- 1 Indicate the value of `sysUpTime` when the last switchover took place.
- 2 Serve as a discontinuity object for MIBs that don't support one, and yet may need one on a switchover.

29.6.2.4 Switchover Notification

To summarize, the `CISCO-RF-MIB` defines a notification to be sent to the management station on a switchover, the `ciscoRFSwactNotif`. This notification serves to:

- 1 Indicate to the management station that a switchover has occurred.
- 2 It serves as a discontinuity notification for management; it indicates to the management station that the counters on the device in question would/may experience discontinuity.

Note Both `coldStart` and `warmStart` notification MUST be suppressed on a switchover.

29.6.3 General Issues Regarding SSO Functionality

This section deals with general issues that are relevant to the SSO functionality of SNMP.

The general issues include:

- Notifications Generated on the Standby
- Tracking Redundancy and Switchover Information
- Issues Not Addressed

29.6.3.1 Notifications Generated on the Standby

In the case of the RPR+ redundancy mode and SSO, the standby RP is fully initialized. From an SSO perspective, this means that the processes on the standby are in the execution phase, as they are on the active. The HA-unaware processes execute as usual on the standby, as they do on the active RP. This may result in their sending notifications to the management station on certain events.

These notifications are prevented from being sent to the drivers because they are dropped as soon as they are seen by the SNMP subsystem. This has two advantages:

- 1 Unnecessary CPU utilization on the standby is avoided. Although the standby is not doing any active processing, this is still unnecessary and would be nice if avoided.
- 2 The network management station would not receive any outdated/incorrect notifications about the device.

All current requests/responses/notifications enqueued on the active SNMP agent are dropped/lost on a switchover to the standby.

29.6.3.2 Tracking Redundancy and Switchover Information

The ENG-61816 documentation, “IOS HA Enhancement Architecture” states that a MIB is required that can provide for monitoring and control information of the redundancy infrastructure and system state. The information that is available from the MIB is:

- 1 A *processor* table that lists all the processors participating in the redundant system. This table includes:
 - (a) The `entPhysicalIndex` for each participating processor: this is the index for this table.
 - (b) The current state of the processor: active or standby. If more than two processors are involved, it borrows from HSRP and indicates “active”, “designated standby”, and “standby”. The “designated standby” is the router that will take over if the active fails, and the remaining routers will then “elect” a new “designated standby”. While there probably is no current design for such a system, from an HA-SSO perspective, provisioning for this in the MIB provides for extensibility.
 - (c) The `sysUpTime` value when the processor entered this state.
 - (d) The length of time (Date&Time syntax) that this processor has been in standby mode.
 - (e) The length of time (Date&Time syntax) that this processor has been in active mode.
- 2 A *history* table that tracks the history of all switchovers that have occurred since the system cold-started. This table contains:
 - (a) A history-index, which is a constantly incrementing value that starts from 1 and wraps around at MAXINT.
 - (b) The `entPhysicalIndex` of the active processor that was shutting down.
 - (c) The `entPhysicalIndex` of the standby processor that was taking over.
 - (d) The reason for the switchover, which could be a text or enumerated value.
 - (e) The timestamp when standby detected the active failure.
 - (f) The timestamp when the active processor indicated re-convergence.
- 3 Global objects to control the size of the history table:
 - (a) An object that determines the maximum number of entries permissible in the history table.

- (b) Alternatively, an object that specifies the maximum time that an entry can live in the history table. The maximum entries value should override the maximum time value.
- 4 The above two tables (and the maximum value limits) MUST persist across SSOs. The rate of sync is “whenever an SSO event occurs” or a new standby processor comes up.
- 5 Notifications indicate:
- (a) A new standby processor is available; when an active processor reloads and comes back up, a “standby” trap is sent out.
 - (b) A switchover event has occurred.

Note Most of the required information is available via the CISCO-RF-MIB. There are some variations from the requirements, though, and these have been listed below.

29.6.3.2.1 The Processor Table

The processor table includes the following objects:

- 1 A new table that lists all processors participating in the redundant system. There will only be two processors: one active and one standby. These are represented by cRFStatusUnitId and cRFStatusPeerUnitId. (Note that these aren’t/might not be the entity indices of the RPs.) And since there is no plan to support more than two RPs in a redundant configuration, this should suffice.
- 2 A new object: failOverTime/redundantSysUpTime, which represents the time when the active processor entered the active state.
- 3 A new object: peerStandbyEntryTime, which is used to represent the time when the standby processor entered the standby state.
- 4 The length of time that a processor has been in active mode can be obtained as a difference between the current value of sysUpTime and the failOverTime. (Note that this is in terms of Time Ticks.)
- 5 The length of time in standby mode can be obtained as a difference between the current value of sysUpTime and peerStandbyEntryTime. (Again, this would be in terms of Time Ticks.)

29.6.3.2.2 The History Table

The CISCO-RF-MIB represents the processors in terms of Unit IDs, cRFStatusUnitId and cRFStatusPeerUnitId, as follows:

cRFStatusPeerUnitId OBJECT-TYPE

SYNTAX RFUnitIdentifier

MAX-ACCESS read-only

STATUS current

DESCRIPTION

“A unique identifier for the redundant peer unit. This identifier is implementation-specific but the method for selecting the ID must remain consistent throughout the redundant system.”

Some example identifiers include: slot ID, physical or logical entity ID, or a unique ID assigned internally by the RF subsystem.”

::= { cRFStatus 3 }

The history table contains information in terms of Unit IDs for the sake of consistency.

The timestamp in the history table is represented as a Date&Time value, since history information is better represented by a Date&Time value rather than a TimeTicks value (the value of `sysUpTime`). The Date&Time value makes sense even across reboots (where `sysUpTime` resets).

Reconvergence time is not considered in the history table because it would require SNMP to coordinate with the routing protocols to figure out when convergence has been achieved.

Regarding global objects in the history table, support for a maximum entry time has not been added.

Regarding notifications in the history table, `ciscoRFProgressionNotif` is generated on active/standby transitions to `standbyCold`, `standbyHot` (in case of standby) and `active` (for the active) states. This indicates the availability of a standby processor.

The information in the `CISCO-RF-MIB` is synced over from the active to the standby (by the redundancy framework subsystem) to present correct and consistent information to the end user on a switchover. The history table named `CRFHistorySwitchOver` that maintains the history of switchovers was added to the `CISCO-RF-MIB`.

The `CRFHistorySwitchOver` history table includes the following objects:

- 1 `CRFHistorySwitchOverIndex`: a monotonically increasing integer for the purpose of indexing the history table. It wraps around to 1 after reaching the maximum value.
- 2 `CRFHistoryPrevActiveUnitId`: indicates the primary unit that went down.
- 3 `CRFHistoryCurrActiveUnitId`: indicates the secondary unit that took over as active.
- 4 `CRFHistorySwitchOverReason`: indicates the reason for the switchover.
- 5 `CRFHistorySwactTime`: indicates the date and the time when switchover occurred.

The following scalar objects are included:

- 1 `CRFHistoryTableMaxLength`: is the maximum number of entries permissible in the history table. A value of zero will result in no history being maintained.
- 2 `CRFHistoryColdStarts`: indicates the number of system cold starts.
- 3 `CRFHistoryStandByAvailTime`: indicates the cumulative time that a standby redundant unit has been available since the last system initialization.
- 4 `CRFStatusFailoverTime`: is the value of `sysUpTime` when the primary redundant unit took over as active.
- 5 `CRFStatusPeerStandByEntryTime`: is the value of `sysUpTime` when the peer redundant unit entered the `standbyHot` state.

Note PRC support for SNMP is planned for Phase 2 development.

29.6.3.3 Issues Not Addressed

This section deals with the issues that are not addressed by the HA MIB sync infrastructure in the current SSO functionality of SNMP.

The general issues include:

- Distributed Management MIBs
- Categorization of Notifications (Not Considered for IOS SNMP SSO)

- Other MIBs

29.6.3.3.1 Distributed Management MIBs

`EVENT-MIB` and `EXPRESSION-MIB` are distributed management MIBs. Neither of these MIBs maintains any information in the startup/running configuration. The persistence of both the `EVENT-MIB` and the `EXPRESSION-MIB` is done via the **write mib-data** CLI command. This would need to be done prior to any syncing operation. Configuration of these MIBs entails configuring quite a few tables and may not be a straight-forward task. The syncing of these MIBs from the active to the standby has been postponed because only a few customers use these MIBs extensively.

A possible work around to sync these MIBs across might be to enable persistence for these MIBs. These should then get synced over as a part of the filesystem sync operation.

29.6.3.3.2 Categorization of Notifications (Not Considered for IOS SNMP SSO)

The PRD requires notifications to be categorized, further suppressed, and co-related. This seems to be an SNMP requirement apart from the HA requirements, since it pertains to general SNMP enhancements, and not exactly to the topic of HA-SSO.

29.6.3.3.3 Other MIBs

There are a few other MIBs that may be considered as candidates for synchronization, but have not been synchronized. Here is a preliminary list of the MIBs that may need synchronization (note that this list may be updated as we come across more MIBs that need to be synchronized):

- 1 `CISCO-CONFIG-COPY-MIB`
- 2 `CISCO-CONFIG-MAN-MIB`
- 3 `CISCO-ENVMON-MIB`
- 4 `CISCO-FLASH-MIB`
- 5 `CISCO-FTP-CLIENT-MIB`
- 6 `CISCO-MEMORY-POOL-MIB`
- 7 `CISCO-ENHANCED-MEMPOOL-MIB`
- 8 `CISCO-PING-MIB`
- 9 `CISCO-PROCESS-MIB`
- 10 `CISCO-ENTITY-FRU-CONTROL-MIB`

29.6.3.4 Memory and Performance Impact

There can be a few processor cycles lost for syncing information across to the secondary. But the benefit obtained from this syncing is a highly available SNMP engine, which requires minimal (if any) reconfiguration to function on a switchover.

29.6.3.5 Packaging Considerations

The SSO support for SNMP is available on all platforms and images that support NSF/SSO.

There is no effect/change in the operation of the SNMP agent on devices not supporting NSF/SSO; it should function as before.

29.6.4 CLI End User Interface

The mode of operation of IOS SNMP is tied to the global redundancy mode of the router, controlled via the redundancy level “mode” command. This means that if the redundancy mode of operation of the router is SSO, then IOS SNMP operates in the SSO mode, or if the redundancy mode of operation of the router is RPR+, then IOS SNMP operates in the RPR+ mode.

In addition, two new command options have been added to two existing commands:

- **debug snmp sync**

The **debug snmp** command has been extended to add the **sync** option. This command can be used to debug SNMP synchronization and failure in synchronization. The standby RP may sometimes be reset as a result of synchronization failures. If the failure occurs when SNMP activities are in progress, this command can be used to verify which activities were in progress, and get more information on the reason for the failure.

- **show redundancy switchover history**

The **show redundancy** command has been extended to add the **switchover history** option. This command displays the switchover history of the device (as also available through the **CISCO-RF-MIB**). It displays the Unit ID of the RP going down, the RP taking over, the time when the switchover occurred, and the reason for the switchover.

29.6.5 Configuration and Restrictions

It should be noted that while the aim of this project is to provide a transparent switchover from the active to the standby, there will always be the possibility of SNMP requests being timed-out/dropped during the switchover window. Further, all the traps in the trap queue of the active RP (going down) might be dropped in the event of a switchover. While the window in which requests go unanswered or traps get dropped is very small (as per the timing restrictions on switchover), there is still a possibility. And the network management applications must be aware of this.

The default redundancy mode of operation of IOS SNMP would be the same as the default global redundancy mode of the router.

One thing to note about the HS-SSO behavior of IOS SNMP is that there is no **warmStart** trap generated on a switchover. The network management stations need to be aware of the fact that the **ciscoRFSwactNotif** indicates that a switchover has occurred, and all counters have been reset. This may require re-tooling the management applications.

29.6.6 Testing Considerations

For a system running with:

- 2 RPs, one active and one standby, with 1 RP in the active mode
- 2 RPs, in split mode/maintenance mode
- 2 different RPs, in active and standby mode, such as RSP2 and RSP4

the following should occur on a switchover:

- **sysUpTime** should not reset across a switchover.
- **sysObjectID** should not change on a switchover.
- **snmpBoots** should stay constant across a switchover.

- discontinuity timestamps in MIBs supporting updation of discontinuity timestamp should be updated.
- `ifIndex` should be maintained across switchovers.
- `entPhysicalIndex` should be maintained across switchovers.

The functionality in the case of the SSO mode should be as specified here; there should be no change in the agent/MIB behavior in a non-SSO mode with respect to the existing behavior.

29.6.7 SSO Application Notification

This section describes how an SSO application finds out about the current operating redundancy mode on an HA system. For example, how an SSO system could continue to support Fast Software Upgrade (FSU), introduced with RPR+, without the need to affect the services being offered by the Active RP. A proposal from Fred Lewis was adopted based on work being done for the 7500.

Briefly, the proposal was that a “gross” version would be associated with an IOS image. The working model was that the Standby would be upgraded with the new version of IOS, then rebooted. This version would be exchanged between the two RPs early on in the initialization sequence of the Standby. If the two IOS versions were not the same, the system would revert from SSO mode back to an RPR+-like operating mode, and stateful communication would be stopped. This was because full versioning (of conversations) would not be introduced until ISSU and a “conservative” means of allowing upgrade without affecting service was required.

The issue was how to tell the stateful (HA-aware) clients that the operating mode had changed and that they should not attempt to continue stateful conversations. There is a requirement for the platform to be able to set the current operating mode, for the clients to be able to retrieve the current operating mode, and for the clients to be informed of a change in the current operating mode. Therefore, the following functions were provided by RF:

1 `rf_set_redundancy_mode()`

Invoked as `reg_invoke_rf_set_redundancy_mode()`. The `rf_set_redundancy_mode()` function is used by the platform to set the value of the `cRFCfgRedundancyMode` and `cRFCfgRedundancyModeDescr` RF MIB objects.

The automatic notification functionality is implemented in `rf_set_redundancy_mode()` via a call to:

```
rf_status_notification(RF_STATUS_REDUNDANCY_MODE_CHANGE, mode);
```

2 `rf_get_redundancy_mode()`

Invoked as `reg_invoke_rf_get_redundancy_mode()`. The `rf_get_redundancy_mode()` function returns the redundancy mode as well as a pointer to the optionally set descriptive string, as described above for `rf_set_redundancy_mode()`.

This functionality provides the support to enable clients to deal with the operational mode switch to support FSU. The `RF_WARM_STANDBY_REDUNDANT` and `RF_HOT_STANDBY_REDUNDANT` values are defined for the mode and are all that is needed for this use.

29.6.8 HA MIB Sync Infrastructure Implementation

The HA MIB Sync Infrastructure implementation was developed in 12.2T; the `/sys/red_facility/rf_registry.regh` file is machine-generated from `rf_registry.reg`:

For example:

```
DEFINE rf_set_redundancy_mode
```

```

*****
* [70]
*
* Invocation API:
* void reg_invoke_rf_set_redundancy_mode(rf_redundancy_mode_e mode,
*                                         const char *description)
*
* Implementation API:
* reg_add_rf_set_redundancy_mode
* reg_delete_rf_set_redundancy_mode
*
* Implemented in: RF common code
*
* Description:
* This routine is to be used by the platform to set the value of the
* RF MIB objects cRFCfgRedundancyMode and cRFCfgRedundancyModeDescr.
* cRFCfgRedundancyModeDescr may be set to NULL if no additional
* mode description text is deemed necessary.
*
* The default setting is crFCfgRedundancyMode = hotStandbyRedundant,
* with no string defined for crFCfgRedundancyModeDescr.
*
* Parameters:
* mode - one of the enumerated values of rf_redundancy_mode_e:
...

```

Note The Redundancy Mode Definitions are described in the [rf_set_redundancy_mode\(\)](#) reference page in the *Cisco IOS API Reference*.

```

...
* ****
STUB
void
rf_redundancy_mode_e mode, \
const char *description
END

DEFINE rf_get_redundancy_mode
/*+*****
* [71]
*
* Invocation API:
* rf_redundancy_mode_e reg_invoke_rf_get_redundancy_mode(char **description)
*
* Implementation API:
* reg_add_rf_get_redundancy_mode
* reg_delete_rf_get_redundancy_mode
*
* Implemented in: RF common code
*
* Description:
* This routine is to be used by the platform to retrieve the value of
* the previously set values of redundancy mode and the associated
* textual description (i.e. those set with a prior call to
* reg_invoke_rf_set_redundancy_mode()).

```

```
*  
* Parameters:  
* none  
*  
* Returns:  
* mode - as defined in rf_set_redundancy_mode() API  
* description - pointer to description string.  
*-----/*-----/  
STUB  
rf_redundancy_mode_e  
char **description  
END  
  
END REGISTRY  
/*
```

29.7 References

The following references are provided for more information on these topics.

- SNMP Notification API
http://wwwin-eng.cisco.com/Eng/IOS/SNMP_WWW/dev-faq.html
- HA MIB Sync Infrastructure
See ENG-126969 at:
[http://wwwin-eng.cisco.com/Eng/IOS/SNMP/snmp_sso_sufts.fm](http://wwwin-eng.cisco.com/Eng/IOS/SNMP/snmp_sso_sufs.fm).
- Gili, P., Colom, J., Jimenez, E., Koushik, K., Swanson, T., “Coruscant Software Functional Specification”, EDCS-363783
- Jimenez, E., Colom, J., Swanson, T., Gili, P., Koushik, K., “Coruscant Software Design Specification”, EDCS-362334
- Colom, J., “Interface Manager Programmer’s Guide”, EDCS-507979
- Banks, D., Donahue, T., “IOS HA Enhancement Architecture”, ENG-61816.
- Baruah, M., Kumar, M., Chakraborty, A., “GSR NSF/SSO SUFS”, ENG-111205.
- Kurnik, T., Berry, B., “CISCO-RF-MIB”.
- Berry, B., “Redundancy Framework Functional Specification”, ENG-47384.
- Lin, M., “CISCO IOS High Availability SNMP Network Management PRD”, EDCS-122444.
- “Integrated Local Management Interface (ILMI) Specification Version 4.0”, af-ilmi-0065.000, ATM Forum.

References

Security

This chapter describes security issues and the security management features available in Cisco IOS.

Note Cisco IOS security development questions can be directed to the secure-ios@cisco.com, ios-infra-security@cisco.com and ios-security@cisco.com aliases.

This chapter includes information on the following topics:

- Secure Coding—To contact experts in this area, use the ios-white-hats@cisco.com alias.
- IOS Security Features

See also section 4.12, “ID Manager” in Chapter 4, “Memory Management” to eliminate the security-related vulnerability in networking devices caused by “freeing twice”.

30.1 Secure Coding

The following tips are provided to help developers write secure code:

- 1 How to Avoid Common Mistakes with Strings
- 2 printf() Pitfall
- 3 IOS Parser and Secure Coding
- 4 strcpy(), strcat(), and sprintf() Limitations
- 5 Zeroing Passwords and Keys
- 6 Asking and Verifying Passwords
- 7 Infinite Loops and MAX Value Usage
- 8 Integer Manipulation Vulnerabilities
- 9 Watching Multiple Queues: Avoiding DOS Attacks FAQ

30.1.1 How to Avoid Common Mistakes with Strings

Be careful when working with strings and character data. C programmers often make a number of mistakes when dealing with strings, due to inexperience or carelessness. These mistakes can have costly consequences, frequently leading to eventual crashes in the program in question, or opening subtle security holes that a malicious agent can exploit to compromise the integrity of a system. Watch for these mistakes, both when developing code and when reviewing code.

These mistakes usually stem from the following root causes:

- Confusing “string” with an “array of characters” (a string is *contained* in a character array, but it doesn’t necessarily wholly fill that array).
- Not properly accounting for the final NUL when working with NUL-terminated strings.
- Assuming all strings are of the NUL-terminated variety, and using functions designed for NUL-terminated strings on strings which are *not* designed for NUL-terminated strings.
- Assuming NUL-terminated outputs from functions that are not guaranteed to produce NUL-terminated results.
- Confusing pointers to character data (`char *`) with the character data itself (the strings) or with the arrays that hold the strings.
- Not initializing pointers to point to something useful.

30.1.1.1 Examples

- 1 Don’t assume storage magically springs into existence just because you have a pointer:

```
char * message;

strcpy(message, "Hello, world"); /* splat! */
```

- 2 Be careful to distinguish strings from the arrays that hold them. For example, with:

```
char buffer[80] = "Hello, world";
```

“buffer” is a character array that is 80 bytes long. The string within it is considerably shorter (12 or 13 bytes, depending on what you count).

In this case, “`sizeof(buffer)`” and “`strlen(buffer)`” both have validity, depending on what you are doing, but they are very different quantities and must not be confused:

```
sizeof(buffer) /* Here the quantity = 80 */
strlen(buffer) /* Here the quantity = 12 */
```

- 3 Similarly, pointers to strings or buffers should not be confused with the strings or buffers themselves. In particular, the size of a pointer to a string is *not* the length of a string, nor the size of the array that holds that string, nor the size of the array needed to hold that string:

```
char * greeting = "Hello, world";
char * message;

message = malloc(sizeof(greeting)); /* wrong */
strcpy(message, greeting); /* splat! */
```

The size of a pointer rarely has any significance when working with strings (on most platforms, it is 4). It is the size of the object pointed to that matters.

- 4 The length of a NUL-terminated string likewise is not the size an array needed to be to hold that string. The following revision of the previous example is closer to being correct, but is still wrong:

```
char * greeting = "Hello, world";
char * message;

message = malloc(strlen(greeting)); /* still wrong */
strcpy(message, greeting);          /* splat! */
```

What is almost certainly intended here is:

```
message = malloc(strlen(greeting) + 1);
```

A more realistic example of this type of error is:

```
char * strdup (const char * old_str)
{
    char * new_str;

    new_str = malloc(strlen(old_str)); /* Allocates N bytes */
    if (new_str) {
        strcpy(new_str, old_str);      /* but copies N+1 */
    }

    return new_str;
}
```

- 5 When dealing with length-related parameters, be careful to distinguish whether they represent the length of strings (trailing NUL not counted), the sizes of strings (trailing NUL included), or the sizes of the buffers holding the strings.

For example, the following code could result in memory corruption:

```
#define MAX_NAME_LENGTH 64

int some_function (char * name)
{
    char name_buf[MAX_NAME_LENGTH];

    if (strlen(name) > MAX_NAME_LENGTH) {
        return STRING_TOO_LONG;
    }

    strcpy(name_buf, name);
    ...
}
```

Either that needs to be:

```
char name_buf[MAX_NAME_LENGTH + 1]
```

or:

```
if (strlen(name) > MAX_NAME_LENGTH - 1) {
```

depending on what MAX_NAME_LENGTH was really meant to represent.

In this particular case, the first choice might be the better one, because if the constant was meant to be the buffersize, a better choice of name (such as NAME_BUFFER_SIZE) would have made that clearer.

6 Avoid passing non NUL-terminated data to functions requiring NUL-terminated inputs:

```
char greeting[] = { 'H', 'e', 'l', 'l', 'o' };
int len;

len = strlen(greeting); /* wrong, may splat */
```

The following is an entirely equivalent form, which is perhaps more likely to get a careless coder into trouble because the initializer may be mistaken as producing a NUL-terminated string:

```
char greeting[5] = "Hello";
int len;

len = strlen(greeting); /* wrong, may splat */
```

In either case, the array `greeting` is five characters long, and does not end in a NUL. When called, `strlen()` will scan memory, looking for a NUL, which means it will scan beyond the end of the array. At the very least, the length returned will be too large. CPU cycles will be wasted scanning memory that wasn't supposed to be scanned. The system may even crash if the scan moves past the end of the memory segment being scanned.

In IOS programming, this type of mistake often occurs when working with textual data in packet buffers, where strings passed in datagrams are not C-style NUL-terminated strings, but the programmer treats them as such:

```
/*
 * The Authentication packet contains the client's name and
 * password, structured as:
 *     Name_len      1 byte
 *     Name          User's name, Name_len bytes long
 *     PW_len        1 byte
 *     Password      User's password, PW_len bytes long.
 */
char * data_ptr;
char * name_buf;
unsigned char name_len;

data_ptr = pak->network_start;
name_len = *data_ptr++;

if (some_debug_flag) {
    buginf("\nUser name is %s", data_ptr); Bad.
}

/*
 * Make a copy of the name while we are at it
 */
name_buf = malloc(name_len);    Bad if NUL-terminator space needed.
strcpy(name_buf, data_ptr);    Bad.
```

7 Beware of unbounded copying of source strings to destinations that are not guaranteed large enough to hold the entire source (including terminating NUL byte, where applicable). For example, even if `inbuf` is a legitimate NUL-terminated string, the following call:

```
strcpy(outbuf, inbuf);
```

will corrupt memory if the array at `outbuf` is not at least as large as `strlen(inbuf)+1`.

Again, failing to account for the NUL terminator byte is one common cause for this type of mistake, but another common cause is simply being careless about the sizes of the buffers being passed around. This is particularly likely to happen when the point where the buffers are created is not near the point where the buffers are used.

```
struct user_context {
    char username[16]; /* Retain name, or a portion of, for printing */
    uint userid;
    ...
};

void save_user_name (user_context *uc, char *name)
{
    strcpy(uc->username, name);
}
```

Whether the above function is safe or not depends on whether or not the callers of the function limit the name to 15 characters.

A particularly insidious and easily overlooked case can occur when modifying code that was previously safe:

```
#define LOGIN_NAME_LENGTH 16

struct login_info {
    char login_name[LOGIN_NAME_LENGTH];
    ...
};

some_function (...)
{
    user_context *uc;
    login_info *login;
    ...
    save_user_name(uc, login->name);
    ...
}
```

Consider if someone later changes `LOGIN_NAME_LENGTH` to 64.

- 8 Beware of functions that may be expected to produce NUL-terminated outputs, but are not guaranteed to do so, for example:

```
strncpy(buffer, string, length);
```

The `strncpy()` function will only produce a NUL-terminated result if the input `string` is shorter than the given `length`.

- 9 `strncpy()` is not a substitute for `strcpy()`.

Programmers sometimes replace calls to `strcpy()` with calls to `strncpy()`, thinking they are making their code better by using a bounded variant of `strcpy()`. This is usually a mistake, `strncpy()` is usually *not* the right function for the job.

Despite its name, `strncpy()` is not really a function for copying strings. It is a memory filling function — a variant of `memset` or `bzero` — that takes an input string for the leading part of the fill pattern. The call:

```
strncpy(buffer, string, length);
```

writes *exactly* `length` bytes into the target `buffer`. These bytes are taken from the string until that string is exhausted, then the remainder of the buffer is filled with zeroes.

Ignoring the previously mentioned fact that the results are not guaranteed to be NUL terminated, `strncpy()` is a very inefficient method to copy a string, unless the string length happens to be very close to the buffer length. Unless the memory fill is a desired affect (for example, deliberately erasing all traces of a password from a buffer), avoid `strncpy()`.

Tip: If your goal is to find a safer variant of `strcpy()`, then a much better choice is the IOS-specific function `sstrncpy()`. This function is just like `strcpy()`, but puts a bound on the amount of data copies and also guarantees a NUL-terminated result.

Note The code comments inaccurately describe `sstrncpy()` as a variant of `strncpy()` instead of `strcpy()`.

10 Beware that the various string-handling functions are not completely consistent in the various development environments. Know which function you are really calling.

Note This depends on the branch and exact level of code you are working with.

For example, consider several of the multiple definitions of `strncpy()` that can be found:

— `boot/util.c`:

```
void strncpy (char * dst, char *src, ulong max)
```

This `strncpy()` is not at all like the ANSI version of `strncpy()`. It does not have the same return value, nor does it fill memory.

— `cisco.comp/ansi/src/strncpy.c`:

```
char * strncpy (char *dst, const char *src, size_t n)
```

This is the IOS standard version, with ANSI-standard semantics.

— `sys/filesys/c7100_ralib_bootflash.c`:

```
#define strncpy sstrncpy
```

This file redefines `strncpy()` to call the Cisco-specific `sstrncpy()` (which does something completely different than what the ANSI C Standard `strncpy()` is supposed to do).

— `sys/rommon/src/strings.c`:

```
char * strncpy (char *dst, const char *src, size_t n)
```

This is the ROMMON version, which like the IOS standard version, is also ANSI-compliant.

Similarly, for `sstrncpy()` you can find:

— cisco.comp/cisco/src/sstrncpy.c:
char * sstrncpy (char *dst, char const *src, unsigned long max)

This is the IOS standard version.

— sys/util/mem_mgr.c: #define sstrncpy strncpy

This file redefines a NUL-terminating, non-filling function, in terms of a non-NUL-terminating, memory-filling function. A good recipe for trouble.

30.1.2 printf() Pitfall

Another common mistake with strings is passing a variable instead of a constant format string to any `printf()` family function. For example:

```
printf(username);
```

where `username` is a string that comes from some external source, such as user input, configuration, or data from packets. While there are some legitimate uses of passing in a carefully controlled variable as the format string, it is a highly discouraged practice in IOS and should be avoided if possible.

In the example above, if `username` contains one or more % signs (that is, `printf()` format specifiers), `printf()` will attempt to access arguments from the `printf()` parameter list that the programmer may not have intended, causing unintended output, security issues, or crashes. For instance, if the user is allowed to enter her `username` above as "%1000000c", `printf()` will access an undefined parameter (because no second parameter is passed to `printf()`) and will attempt to output that character one million times. If the `username` typed in is "%s", the example will often cause a crash.

Instead, the programmer should have used the following:

```
printf("%s", username);
```

Note This pitfall applies to any function from the `printf()` family, such as `sprintf()`, `fprintf()`, and `snprintf()`, as well as any Cisco-defined function that takes a format string and arguments.

30.1.2.1 printf(), snprintf(), and buginf() Format Strings

Compilers detect format string errors if the actual formal string is passed as an argument. However, it will not detect errors if the format string is a character pointer.

For example:

```
1 char *formatstr = "%s %d\n";
2 char *str = "Hello world";
3
4 int main(void)
5 {
6 printf(formatstr, str);
7 printf("%s %d\n", str);
8 return (0);
9 }
/router/bin/gcc.c2.95.3-p8 -Wformat -g 25.c
25.c: In function `main':
25.c:7: warning: too few arguments for format
infra-view7:206>
```

The compiler will be able to detect the print format string error in line 7 but will not be able to detect it in line 6. To avoid this issue, we need to provide the actual format string in `printf()`. Even if the string is replicated, compilers are intelligent enough to make it appear once in `rodata`, thereby avoiding `rodata` bloat.

30.1.2.2 Recommendation: Use `puts()`

If you want to output a raw string with IOS, use `puts()` instead of `printf()`. For example, use `puts()` instead of `printf()` in these cases:

- 1 When you have no format specifiers (that is, % signs) in your string that you want interpreted. `printf()` is a relatively CPU-intensive function. If you do not need your format string interpreted, do not use the CPU doing unnecessary work.
- 2 When you specifically want to avoid interpretation of format specifiers because your string *might* contain them. For example, using `puts()` would have been another correct way to have solved the `printf(username)` bug from above. For instance:

```
puts(username);
```

is a valid solution.

Note The ANSI `puts()` will output the string followed by a newline, but not the IOS `puts()`. Instead, the IOS `puts()` behaves like the ANSI `fputs()` but without the filespec.

In IOS we often want to output a newline followed by a string (which does not include a newline).

As shown in the example from the rainier branch below, the IOS `puts()` outputs newlines exactly/only as they are present in the parameter string:

```
/*
 * puts
 * Print a character string on the primary output
 */

void puts (char *s)
{
    char *n = NULL;

    if (stdio) {
        for(;*s;) {
            doprintc(&n, stdio, *s++);
        }
    } else {
        errmsg(&msgsym(BADPRINT,SYS), "puts");
    }
}
```

30.1.3 IOS Parser and Secure Coding

Historically, buffer overflows were not considered during parser processing. However, current challenges include the major concern of unbounded string input, because we know the receiving buffer is bounded. To improve the security of IOS, switch to the following macros, which provide a maximum number of bytes to parse:

- Use the bounded `STRING_MAX` in place of the unbounded `STRING`.
- Use the bounded `GENERAL_STRING_MAX` in place of `GENERAL_STRING`.

30.1.4 `strcpy()`, `strcat()`, and `sprintf()` Limitations

The `strcpy()`, `strcat()`, and `sprintf()` functions do not have boundary checks, which can cause buffer overflow issues if a bigger string is copied onto a smaller buffer. We need to use a variant of these functions that will copy strings into a buffer for a specified buffersize. In the case of string copy, we need to use `sstrncpy()`:

```
char *sstrncpy(char *dst, const char *src, size_t length)
char *strncat(char *dst, const char *src, size_t length)
int snprintf(char *cp, size_t length, const char *fmt, ...)
```

Note For more information on significant differences between `sstrncpy()` and other functions, see the Usage Guidelines in the [sstrncpy\(\)](#) reference page.

In the case of `strncpy()`, the `NULL(' \0 ')` character is not copied to the end of the buffer if the length of `src` is greater than the length of `dst`. So an explicit `NULL` assignment to the end of the buffer needs to be done to avoid a non-null terminated string.

For example:

```
#define STRING_NAME_LEN 11
char *src = "Hello World"; /* length of this string = 11 */
char dst[STRING_NAME_LEN];
strncpy(dst, src, STRING_NAME_LEN);
```

This will result in `dst[STRING_NAME_LEN - 1]` to be “d.” There will not be NULL at the end of the string.

To make sure the string ends in NULL, the recommended array definition should be `STRING_NAME_LEN + 1` and there should be an explicit NULL written to end of the string.

The above example should be:

```
#define STRING_NAME_LEN 11
char *src = "Hello World"; /* length of this string = 11 */
char dst[STRING_NAME_LEN + 1];
strncpy(dst, src, STRING_NAME_LEN);
dst[STRING_NAME_LEN] = '\0';
```

We should use Cisco’s (not ANSI) safe version of `strncpy()` to make the string NULL terminated to avoid explicit NULL termination:

```
char *sstrncpy (char *dst, char const *src, unsigned long length)
```

The above example here would be:

```
#define STRING_NAME_LEN 11
char *src = "Hello World"; /* length of this string = 11 */
char dst[STRING_NAME_LEN + 1];
sstrncpy(dst, src, STRING_NAME_LEN + 1);
```

In the above example, we make sure that we pass the entire length of `dst` to make sure that it is NULL terminated. In addition, if there are remaining bytes after the string is copied, `strncpy()` will copy the NULL character to the remaining destination buffer. You should make sure that an arbitrary length is not passed as an argument. `sstrncpy()` should be used always to avoid these issues and `sstrncpy()` has better performance when compared to `strncpy()`.

`static_iosbranch` should catch errors related to `strncpy()` where a NULL character is not terminated. The `strncat()` function copies the NULL character to the end of the string and the length we pass does not account for the NULL character. This would mean the length of data copied from `dst` to `src` will be the length from `src` and a NULL.

For example:

```
#define STRING_NAME_LEN 11
char *src = "World"; /* length of this string = 5 */
char dst[STRING_NAME_LEN + 1];
size_t len = STRING_NAME_LEN;
strncpy(dst, "Hello ", STRING_NAME_LEN);
dst[STRING_NAME_LEN] = NULL;
len -= strlen(dst); /* len will be 5 */
strncat(dst, src, len);
```

`snprintf()` makes sure that a NULL is terminated at the end of the string:

```
#define STRING_NAME_LEN 11
char *src = "Hello World"; /* length of this string = 11 */
char dst[STRING_NAME_LEN + 1];
snprintf(dst, STRING_NAME_LEN + 1, "%s", src);
```

dst[STRING_NAME_LEN] will be made zero by snprintf().

30.1.5 Zeroing Passwords and Keys

Whenever a password is stored in a stack or in heap, your application should make sure that it is zeroed before being freed or coming out of the function stack.

An example would be in ntp_config.c (in the ntp_global_config_cmd() routine):

```
case NTP_AUTH_KEY_CMD:  
/*  
 * ntp authentication-key ## md5 value [cryptotype]  
 * no ntp authentication-key ##  
 *  
 * OBJ(int,1) = key number  
 * OBJ(int,2) = value encryption type  
 * OBJ(int,3) = TRUE if cryptotype present - not available to user.  
 * OBJ(string,1) = key text  
 */  
if (csb->sense) { /* positive form */  
    MD5auth_setkey(GETOBJ(int,1), GETOBJ(string,1),  
    GETOBJ(int,3),  
    GETOBJ(int,2),  
    csb->resolvemethod == RES_MANUAL ? TRUE : 0);  
} else {
```

The above function does not zero the md5 authentication string and key number after it is received from csb. Note that the parser takes care of zeroing the csb when the next command is entered, and so it is possible for someone to dump the csb contents through another EXEC terminal. Therefore, it is always good to make sure that your application can zero out the contents of csb, which will contain the keys and passwords.

There are cases where the compiler stores the key onto the stack if it is an integer or the program stores the key onto the stack for temporary use.

An example here would be the MD5Transform routine in util/md5.c:

```

static void
MD5Transform (
    unsigned long int *state,
    unsigned char *block)
{
    unsigned long int a = state[0], b = state[1], c = state[2], d = s
    tate[3],
    x[16];
    .....
    .....
    .....
    II(a, b, c, d, x[4], S41, 0xf7537e82); /* 61 */
    II(d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
    II(c, d, a, b, x[2], S43, 0x2ad7d2bb); /* 63 */
    II(b, c, d, a, x[9], S44, 0xeb86d391); /* 64 */
    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
    /*
     * Zeroize sensitive information.
     */
    MD5_memset((unsigned char* ) x, 0, sizeof(x));
}

```

The code makes sure that array `x`, which is allocated on the stack, is zeroed before returning from this routine.

Also, your application should make sure that the keys that are passed in as arguments or keys stored to a local variable are zeroed. Because these variables resides on the stack, it is possible that the stack is not overwritten by another function, which would mean that someone can dump the stack and decode the key values.

30.1.6 Asking and Verifying Passwords

Two functions are used to handle the password entry. They are:

- `askstring()`—`boolean askstring (tt_soc *tty, char *buffer, int length, char *prompt, int timeout, int retries, boolean noecho)`
- `askpassword()`—`boolean askpassword (tt_soc *tty, char *target, uint encryption)`

These two functions allow the system to prompt for password entry. They also handle timeouts, retries and password verification.

30.1.7 Infinite Loops and MAX Value Usage

A comparison of an (un)signed number to a maximum value can cause infinite loops. For example, in the `ipc_yank_older_messages()` routine, a message needs to be searched from the current message to the maximum possible number of messages. In addition, there is a condition where the index value overflows but never reaches the loop break condition:

```
/*
 * Before Wrap
 */
for (index = next_ack; index <= IPC_MAX_SEQUENCE; index++) {
    if (!ipc_process_cum_ack(hdr, ack_seatid, dest_port_info, index) &&
        ipc_debug_acks) {
        buginf("\nMSG NOT IN TABLE! (la = %d, na = %d, ca = %d, "
              "ia = %d)", last_ack, next_ack, index, curr_ack);
    }
}
```

`IPC_MAX_SEQUENCE` is `0xffff`. The code will run in an infinite loop because the index that is unsigned short will grow to `0xffff` and then fallback to zero and continue. These problems are identified using `static_iosbranch` routines.

Always make sure of the boundary conditions when comparing with MAXIMUM values of integers or short.

Another example related to MAXIMUM value being used is in the allocation of memory or traversing an array:

```
if ((oid_ptr = MakeOID(NULL, MIN(length+2,MAX_OID_SIZE))) == NULL) {
    DPRINTF((APWARN, "ParseOID, oid_ptr MakeOID\n"));
    return (NULL);
```

Here if the value of `length` is `MAXUSHORT`, we will end up allocating less memory, which can cause corruption. Running `static_iosbranch` on the changed code should catch these issues.

30.1.8 Integer Manipulation Vulnerabilities

There are three main integer manipulations that can lead to security vulnerabilities:

- Overflow and Underflow
- Signed versus Unsigned Errors
- Truncation Errors

On their own, these issues may not cause security errors. However, if your code exhibits one or more of these issues and manipulates memory, the potential for a buffer overrun error or application failure increases.

Advice on how to prevent such errors is provided in the following sections:

- Remedies Using Unsigned Integers
- Key Code Reviewing Points

30.1.8.1 Overflow and Underflow

What's wrong with this code?

```
bool func(size_t cbSize) {
    if (cbSize < 1024) {
        // we never deal with a string trailing null
        char *buf = (char*)malloc(cbSize-1);
        memset(buf, 0, cbSize-1);

        // do stuff

        free((void*)buf);

        return true;
    } else {
        return false;
    }
}
```

The code is correct, right? It validates that `cbSize` is no larger than 1 KB, and `malloc` should always allocate 1 KB correctly, right? Let's ignore for the moment the fact that the return value of `new` or `malloc` should be checked. Also, `cbSize` cannot be a negative number, because it's a `size_t`. But what if `cbSize` is zero? Look at the code that allocates the buffer: it subtracts one from the buffer size request. Subtracting one from zero causes a `size_t` variable, which is an unsigned integer, to wrap under to `0xFFFFFFFF` (assuming a 32-bit value), or 4 GB. Your application just died, or worse!

30.1.8.2 Signed versus Unsigned Errors

Take a quick look at the following code. It's similar to the example in section 30.1.8.1 "Overflow and Underflow." See if you can spot the error, and if you do, try to determine what the result is.

```
bool func(char *s1, int len1,
          char *s2, int len2) {

    char buf[128];

    if (1 + len1 + len2 > 128)
        return false;

    if (buf) {
        strncpy(buf, s1, len1);
        strncat(buf, s2, len2);
    }

    return true;
}
```

The problem here is that the string sizes are stored as signed integers, so `len1` can be larger than 128 as long as `len2` is negative, hence the sum is less than 128 bytes. However, a call to `strncpy()` will overflow the `buf` buffer.

30.1.8.3 Truncation Errors

Now let's look at the last attack type, by way of another code example:

```
bool func(byte *name, unsigned long cbBuf) {
    unsigned short cbCalculatedBufSize = cbBuf;
    byte *buf = (byte*)malloc(cbCalculatedBufSize);
    if (buf) {
        memcpy(buf, name, cbBuf);
        // do stuff with buf
        if (buf) free(buf);
        return true;
    }

    return false;
}
```

What if `cbBuf` is `0x00010020`? `cbCalculatedBufSize` is only `0x20` because only the lower 16-bits from `0x00010020` are copied. Hence only `0x20` bytes are allocated, and `0x00010020` bytes are copied into the newly allocated target buffer. Notice that compiling this code with a setting of `/W4` yields this typical warning:

```
C4244: 'initializing' : conversion from 'unsigned long' to 'unsigned short',
possible loss of data
```

Note On architectures where the size of `short`, `int`, and `size_t` are all the same and `long` is longer, you'll likely get incorrect results but the allocation and copy will both be of the smaller size. On the few architectures where `short`, `int`, `long`, and `size_t` are all the same size, you'll get the correct behavior. For IOS, `short` is smaller than the other types, so these don't apply to IOS.

30.1.8.4 Remedies Using Unsigned Integers

- 1 Who would have thought that simply manipulating integers could lead to security problems? A simple remedy to vulnerable code like this:

```
if (A + B > MAX) return -1;
```

is to use this code with unsigned integers:

```
if (A + B >= A && A + B < MAX) {
    // cool!
}
```

The first operation `A + B >= A`, checks for the wrap-around, and the second makes sure the sum is less than the target size.

- 2 Use unsigned integers (such as `unsigned long` and `size_t`) for array indexes, buffer sizes, and the like.

Note Use `size_t` for the size of anything internal to a specific architecture. For data which is shared externally, pick an appropriate type for all architectures that may use the data. That is, if you have a 32-bit processor and a 64-bit processor, and you are sending a stream that starts with the length of the stream, the 32-bit processor could write more than 2^{32} bytes into an array on the 64-bit processor, so `unsigned long long int` may be more appropriate.

Note While you can use a `long int` on a given processor with one compiler and set of compiler flags and always have it work, you can't pass a 64-bit `long int` from a 64-bit mode MIPS environment to a 32-bit PowerPC environment which is expecting a 32-bit `long int`. The 1999 language standard allows for the definition of a set of common exact types like `int32_t`, which can be used on code that is compiled on more than one architecture for passing data between two or more architectures.

30.1.8.5 Key Code Reviewing Points

Keep the following things in mind when compiling or reviewing code for integer-related issues:

- Compile your code with the highest possible warning level: `/W4`.
- Use `size_t` or `unsigned long` for buffer sizes and element counts. There is no reason to use a signed value for such constructs. Keep in mind that a `size_t` is a different type depending on the platform you use. A `size_t` is the size of a memory address, so it is a 32-bit value on a 32-bit platform but a 64-bit value on a 64-bit platform.
- If your code performs any kind of integer manipulation (addition, multiplication, and so on) where the result is used to index into an array or calculate a buffer size, make sure the operands fall into a small and well-understood range. Be wary of signed arguments to memory allocation (functions such as `malloc` and so on) because they are treated as unsigned integers.
- Watch out for operations that yield C4018, C4389, and C4244 warnings.
- Watch out for casts that cast away the C4018, C4389, and C4244 warnings.
- Investigate all use of `#pragma warning(disable, Cnnnn)` that disable the C4018, C4389, and C4244 warnings. In fact, comment them out, re-compile, and check all new integer-related warnings. Code migrated from other platforms or compilers may make different assumptions about data sizes. Watch out!

30.1.9 Watching Multiple Queues: Avoiding DOS Attacks FAQ

Question

Is there a limit to the number of queues that a process can watch? Even if there is no theoretical limit, are there any rules of thumb that say that "if you're watching too many queues, then such-and-such bad things will happen?"

For example, in my application, I might end up watching several hundreds of queues.

All queues contain the same kind of data, namely packets (or a small part of the packet header). The device my code will run on is a metro ether switch connected to residential subscribers. Each subscriber sends ethernet data to the switch. On the switch, the source MAC address of every data frame is examined. If it is a new (unknown) MAC address, the packet is enqueued to my process.

If it is an old (known) MAC address, it is forwarded in the hardware, and I am not concerned with it. Now, for enqueueing the packet data to my process, there are two approaches. I can have a single queue into which packets coming from all subscribers is enqueued. Or I can have one queue per subscriber.

Suppose I have a single shared queue. Suppose one subscriber is doing something malicious, and floods the switch with packets with random source MAC addresses, then the queue will be filled with packets coming from only this subscriber. Packets coming from other subscribers will not get a chance to get enqueued (since the queue is of finite depth). Effectively this denies service to all the

other subscribers. So the easy solution is to have one queue per subscriber. And I have to ensure that my process is fair enough to look at all the queues instead of just one queue. And since there can be hundreds of subscribers per switch, I am talking about hundreds of queues.

Under normal conditions, most of these queues would be empty, but if the box is the target of malicious activity (a DOS attack), one or more of these queues would get full. What should I watch out for when processing such large numbers of queues?

Answer

The recommendation is to make use of *one queue only* and address the DOS attack by doing something similar to TCP's RED (Random Early Discard). Normally when the queue starts to get large, everything is accepted until the queue limit, then RED discards everything until the queue shrinks. RED (and weighted RED) start discarding entries to the queue before it hits its limit, where the probability of discard is a function of the queue length. The benefit of RED (under DOS attack conditions) is that the probability of accepting a message is about the same as without RED, but once something is accepted, the delay in the queue is far less. This results in a better service level during the DOS attack. Under normal conditions, the queue will be short, so the probability of discarding a message is low.

Also, if you know that one interface is generating most of the requests, you could modify the RED algorithm to increase the probability of discarding a message from that interface (before you even attempt to queue the message) One approach is to:

- Use a single queue
- Enforce a per-subscriber quota for the number of items that the specific subscriber has in the queue
- The quota can be a fixed number or can depend on the current queue depth, etc.

30.2 IOS Security Features

30.2.1 AutoSecure

This section describes the functions and data structures that provide an interface for the Auto-Security feature, or AutoSecure for short, in the Cisco IOS software. AutoSecure was added in 12.3.

30.2.1.1 The AutoSecure API

The AutoSecure API includes the following functions:

- `autosec_platform_set_value()`
- `reg_invoke_autosec_add_module()`
- `reg_invoke_autosec_finish()`
- `reg_invoke_autosec_interface()`
- `reg_invoke_autosec_show_int_config()`

See the API reference pages for more details.

30.2.1.2 AutoSecure Management

The AutoSecure feature is managed primarily through the Cisco IOS CLI. There are two main Exec commands. They are:

- **auto secure [management | forwarding] [no-interact]**

The **auto secure** command performs security configuration of the management and/or security planes of the router. The **no-interact** option performs only that part of the configuration that can be done without any interaction.

The result is that the command applies the configuration to running-configuration. If the configuration is interactive (which is the default), it prompts the user for confirmation to save the configuration. It also displays the configuration generated after security processing before saving the configuration to running-configuration.

- **show auto secure config**

This command displays the configuration created as part of the previous run of the AutoSecure feature the last time it was performed. Since AutoSecure does not store the state across reloads, this command will not show the AutoSecure configuration that was done before last reload.

There are two **config** CLI commands:

- **security passwords min-length *length***

This command ensures that all the passwords that are configured from this point forward are longer than *length* in length. The passwords currently taken into consideration are user passwords, enable passwords and secrets, and line passwords. After configuring this command, any password that is configured with a length less than *length* will fail with an error. For example:

```
ios210(config)# security passwords min-length 6
ios210(config)# enable password lab
Password too short - must be at least 6 characters. Password not configured
```

- **security authentication failure rate *threshold-rate***

This command ensures that if there are continuous failures in attempting to access the device and the number of failures is greater than the configured limit in the last minute, it will generate a **syslog** message. If the number of unsuccessful login attempts in the last minute for the device exceeds *threshold-rate*, it generates a **syslog** message. For example:

```
ios210(config)# security authentication failure rate 10
```

30.2.1.3 The AutoSecure Subsystem

The AutoSecure subsystem provides centralized infrastructure that can be extended for features in the future. Currently, it supports the set of features mentioned in the AutoSecure Functional Specification, document number EDCS-211213.

Every component/subsystem that wants to be part of AutoSecure must register with the AutoSecure subsystem in their initialization routine using the `reg_invoke_autosec_add_module()` function. The component must provide a function that performs the component's specific security configuration on behalf of AutoSecure, and a function which displays that configuration. The component must also provide a sequence ID that it must define in `sys/ui/auto_secure.h`. The sequence ID aids in calling the supported AutoSecure modules in a particular defined order rather than arbitrarily. If the component has order needs, it must choose its sequence ID appropriately and should document those needs in a comment to prevent inadvertent misordering.

Each AutoSecure module or component will have a configuration template that is to be added to the running-configuration. The template might change, depending on the current configuration and user interaction. This configuration will be added to the AutoSecure configuration buffer; at the end of a successful AutoSecure session, meaning the user did not quit with `^c` and there are no resource errors, etc., this configuration will be added to the running-configuration.

30.2.2 How to Use Private VRFs to Isolate Interfaces

This section describes a way to use a private VRF (VPN (Virtual Private Network) Routing/Forwarding) to either hide internal IP address information from the user, or make a management-only interface process traffic to and from the box without being a transit interface. Although these are two independent implementations, the approach to using a private VRF to isolate both internal-only and externally accessible management interfaces is the same.

“Internal IP address communication” refers to the use of IP addresses for different modules inside of a box. Usually, the 127.x.x.x network is used. A “management-only interface” is typically an out-of-band management interface that should not be allowed to act as a packet data transit interface for traffic (such as an Ethernet port on the GRP of a GSR or a FastEthernet port of a NSE-100 on a 7304).

In this section, “30.2.2.2 Guidelines for Using a Private VRF Inside the Box” outlines the general guidelines for using private VRFs to isolate interfaces, and “30.2.2.3 Example of Using a VPN to Hide the Internal Network” provides an example implementation of using a private VRF to hide the internal IP information used for communication between modules inside the box.

A similar approach to the approach in “Example of Using a VPN to Hide the Internal Network” could also be used for the management-only interface issue.

30.2.2.1 Terms

MCEF

Multi CEF (Cisco Express Forwarding). CEF running with multiple forwarding FIBs (Forwarding Information Base) as part of VPN configurations.

NRP

Node Route Processor

NSP

Node Switch Processor

out-of-band

Transmission using frequencies or channels outside the frequencies or channels normally used for information transfer.

PAM MBOX

Port Adapter Module Mail Box. The shared memory space between NSP and NRP that is used as a communication path.

PAM MBOX interface

The pmbox IDB interface on top of PAM MBOX capable of forwarding IP traffic.

VRF

VPN Routing/Forwarding. The mechanism to configure multiple routing tables and MCEF functionality.

VRFifying

Slang used to describe the software task of enhancing a protocol or application to work with MCEF technology.

30.2.2.2 Guidelines for Using a Private VRF Inside the Box

Using a private VRF to handle IP address space inside the box allows you to prevent all types of routing through an interface while still allowing locally originated packets to be sent through the interface. And it is transparent to the user, something under the covers that the user wouldn't know about; it wouldn't show up in any **vrf show** commands.

12.0(5)T introduced VPN with the interface-specific routing tables that are used to provide this functionality. The VPN configuration prevents exposing any private addresses used in the internal network to the network outside the system/box. Since the private network addresses are in separate routing tables and not in the Global Routing Table, outside users/networks can never see the system-internal network.

To prevent routing to the interface, you must use the `socket_set_option()` function with the `SO_VRFTABLEID` option to set the VRF routing table ID.

Caution There are security issues with using a VRF to handle IP address space inside the box when packets get sent to the internal addresses and there are CLI collisions with the VRF names. To avoid these issues, be sure to follow the guidelines in this section for using a private VRF to handle IP address space inside the box.

The general guidelines for using a private VRF inside the box are as follows:

Step 1 Put the interface under a VRF.

Step 2 Provide a mechanism to populate routes into the VRF and force locally generated packets to use the VRF.

For an example of how to implement these guidelines, see “Example of Using a VPN to Hide the Internal Network”

30.2.2.3 Example of Using a VPN to Hide the Internal Network

This implementation provides a single control point for the box and provides isolation between the user data flow and system management data. This is accomplished by having NRP (or your platform’s equivalent) managed and communicated to from the NSP (or your platform’s equivalent) over an internal IP network inside the box, and by having a VPN used to hide the internal network from outside the box.

The VPN runs on the NRP (encompassing the PAM MBOX interface) and on the NSP (encompassing just the PAM MBOX interface there). The PAM MBOX interfaces provide a point-to-point communication link between the NSP and multiple NRPs, thus we have separate networks inside the NSP.

This implementation requires the following:

- 1 Automated Configuration of the Interfaces
- 2 NSP pmbox Interface Creation

The implementation described in this section is intended to be generic and platform-independent; however, it is based on the platform-specific implementation used to prevent routing to a Cisco 6400 system described in ENG-46442.

Note Be aware that your platform may need a significantly different implementation for the NRP equivalent and the NSP equivalent.

30.2.2.3.1 Automated Configuration of the Interfaces

The functional operation of the private interface is critical to operation. The private interface is statically configured at system boot time or when an NRP is inserted. Plan to make the internal network statically configured, with no option for the end user to modify the configuration.

Private Network Layout

PAM MBOX interfaces (`pmbox`) are modeled after point-to-point interfaces. From the NSP perspective, each `pmbox` will be on a separate network. Use the globally available `10.x.x.x` network as the basis for these networks.

For example, the network addressing can be as follows:

`10.slot.Y.X`

where:

`X = 1` for NSP

`X = 2` for NRP

and:

`Y = 0` for the upper subslot

`Y = 1` for the lower subslot (During normal operation we will never see `Y = 1`. It is reserved for potential use during system bring-up.)

Also, the subnet mask can be:

`255.255.255.0`

Thus, with an NRP in slot 2, the NSP `pmbox` interface will be configured with IP address `10.2.0.1` and on the NRP, the `pmbox` will be configured with IP address `10.2.0.2`.

With an NRP in slot 6, the NSP `pmbox` interface will be configured with IP address `10.6.0.1` and on the NRP, the `pmbox` will be configured with IP address `10.6.0.2`.

Auto-Configuration

In order for NRPs to be managed by the NSP at boot-up or at NRP insertion, configuration for system components must be automatic. An unconfigured box must be able to be controlled/monitored over the `pmbox` interface. In addition, you must not allow the user to remove necessary configuration elements. For example, if the user downloads into NVRAM a config without the necessary elements, you still need to operate.

To establish auto-configuration, configure the following elements for both the NSP and the NRP:

- System Configuration
- CEF Switching
- VRF Definitions
- Private Interface Configuration

Details on establishing auto-configuration for these elements are provided in the next subsections.

System Configuration

Due to the simplified nature of the private network, no dynamic/static route entries need to be established for the interfaces. Entries will be automatically instantiated as a result of an operational interface.

CEF Switching

CEF switching is required by VPNs. At system power-up time, both the NSP and NRP will automatically enable CEF switching. The CEF action routine must be modified with a registry call so that images that are based on the NSP/NRP will not be able to disable CEF switching. Therefore, the PAM MBOX interface driver must support CEF switching.

VRF Definitions

The VRF configuration will define a VPN for private use inside the system. In addition to defining the `vrf` instance, you will define the route distinguisher (`rd`) for that `vrf`. In practice, since you are not carrying VPN data through a network, the `rd` values will not be used. Enabling the `rd` field will, though, enable correct `verve` functionality on the interfaces.

The “system-private” `verve` is created to support the internal network, and an `rd` of `0 : 0` is used.

It is desirable to prevent the user from removing this configuration as well. The goal is to modify the action routine associated with removing `vrf`s and via a registry call, and prevent the user from removing this internal network.

Private Interface Configuration

Each interface associated with a `vpn` must be attached to a `vrf`. When the interface is created, a `vrf` will be attached to it. When the `vrf` command removes the existing IP address, you will (re)assign the IP address to the interface.

As mentioned previously, the IP address used will be based on the slot in which the NRP is present. To prevent the user from modifying the `pmbox` interface and because this interface is *only* used for internal data, you will need to make this interface unconfigurable, except at interface creation time. It is the responsibility of the `pmbox` interface to enforce this functionality.

Configuration Events

The above events correspond to specific actions performed when specific system events occur. The NSP power-on/Failover events and actions control VPN configuration.

NSP Power-on/Failover

At power-on, the NSP will auto-configure CEF switching and create the `vrf` definition. Note that in addition to configuring CEF on the NSP, this auto-configuration will also set up host aliases for each NRP in the system. This will make it easier for users to `telnet` to each `nrp2`.

For example, the following script is executed in the platform-specific code prior to interface creation:

```
ip cef
ip vrf system-private
rd 0:0
exit
alias exec nrp1 nrps1 telnet /vrf system-private 10.1.0.1
alias exec nrp2 nrps2 telnet /vrf system-private 10.1.0.2
alias exec nrp3 nrps3 telnet /vrf system-private 10.1.0.3
alias exec nrp4 nrps4 telnet /vrf system-private 10.1.0.4
alias exec nrp5 nrps5 telnet /vrf system-private 10.1.0.5
alias exec nrp6 nrps6 telnet /vrf system-private 10.1.0.6
alias exec nrp7 nrps7 telnet /vrf system-private 10.1.0.7
alias exec nrp8 nrps8 telnet /vrf system-private 10.1.0.8
end
```

The **alias** command will allow a user to connect to a particular **nrp2** with ease.

nrps1 will telnet to the **nrp2** in slot 1.

30.2.2.3.2 NSP pmbox Interface Creation

At the time the **pmbox** interface is created for the system, the IP address will be calculated and the VRF will be attached to this interface.

For example, the following script is executed via the driver code:

```
int pmbox x/y/0
  ip vrf forwarding system-private
  ip address 10.X.Y.Z 255.255.255.0
  exit
end
```

NRP Power-on/Failover

When the NRP is powered on, configuration is applied to the platform in two places. First in the initialization code in platform-specific code, **cef** will be enabled and the **vrf** created (similar to what happens to the NSP).

For example, the following script is executed in platform-specific code prior to interface creation:

```
ip cef
  ip vrf system-private
  rd 0:0
  exit
end
```

Thereafter, when the **pmbox** interface is created for the system, the private IP address will be calculated and the **pmbox** interface will be attached to the **vrf** and configured with the IP address.

For example, the following script is executed via the driver code after the **ldb** has been advertised to IOS:

```
int pmbox 0/0/0
  ip vrf forwarding system-private
  ip address 10.X.Y.Z 255.255.255.0
  exit
end
```

And, drivers and system code can use the following function to configure the interfaces:

```
ifs_copy_buffer_to_file(ifs_running_config_name,
                        command_buffer, /* command_buffer
                        contains a pointer to the string that you wish to introduce to the parser */
                        strlen(command_buffer),
                        FALSE);
```

It is presumed that when a card is removed and reinserted, the OIR code will retain the correct VPN configuration. Thus, these configuration actions need not occur when an NRP is reinserted into an operation platform. (The NRP that is inserted, though, will go through a full boot and thus will need to reinstate the configurations).

Applications Communication Over the pmbox Interface

Different data streams flow over the PAM Mail Box: some on top of IP over the pmbox interface and some over the raw PAM Mail Box interface. Because of the nature of the VPN interface, those protocols which flow over the pmbox interface must be VRFified.

The following data streams flow over the PAM Mail Box:

- SNMP:
 - NSP - SNMP relay (needs VRFifying)
 - NRP - SNMP client (needs VRFifying)
- Telnet: (already VRFified)
- Syslogging:
 - NSP - syslogd (needs VRFifying)
 - NRP - syslog (needs VRFifying)

The data streams that flow over the raw PAM Mail Box interface (where no VPN knowledge is necessary) are as follows:

- Image Download
- IF Console
- NRP Health Monitoring
- ROMMON Variables
- Configuration Download

AAA

31.1 Overview

This chapter describes authentication, authorization and accounting in Cisco IOS as of 12.4(11)T.

31.2 AAA Functions

The following sections describe the AAA functions:

- AAA Attribute Functions
- AAA Utility Functions
- Authentication and Authorization Functions
- AAA Profile Functions
- Server Group Functions
- AAA Accounting Functions

31.2.1 AAA Attribute Functions

Table 31-1 describes the AAA attribute functions

Table 31-1 AAA Attribute Functions

Function	Description
<code>aaa_attr_add()</code>	Constructs an attribute and append it to the requested list
<code>aaa_attr_add_string()</code>	Adds a an attribute of type <code>string</code> to a list
<code>aaa_attr_add_ulong()</code>	Adds an attribute of type <code>ulong</code> to a list
<code>aaa_attr_cursor_valid()</code>	Validates that the cursor points to an existing attribute in a list
<code>aaa_attr_default_handler()</code>	Checks whether the client has passed a valid cursor
<code>aaa_attr_delete()</code>	Deletes an attribute
<code>aaa_attr_event_add()</code>	Adds the event attributes
<code>aaa_attr_event_add_binary()</code>	Adds a binary attribute to a list

AAA Functions

Table 31-1 AAA Attribute Functions (continued)

Function	Description
<code>aaa_attr_event_add_string()</code>	Adds a string type attribute to a list
<code>aaa_attr_event_add_ulong()</code>	Adds an event attribute of type <code>ulong</code> to a list
<code>aaa_attr_get_by_type()</code>	Gets the next attribute type
<code>aaa_attr_get_enum_name_from_type()</code>	Gets the human-readable form from a AAA attribute type and a value
<code>aaa_attr_get_format_from_type()</code>	Gets the attribute format from the attribute type
<code>aaa_attr_get_length()</code>	Gets the length of the attribute pointed to by the cursor
<code>aaa_attr_get_name()</code>	Gets the name of the attribute
<code>aaa_attr_get_name_from_type()</code>	Gets the attribute name from its type
<code>aaa_attr_get_next()</code>	Gets the next attribute
<code>aaa_attr_get_protocol_type()</code>	Retrieves the protocol type of a AAA attribute list
<code>aaa_attr_get_tag()</code>	Gets the next tag of the attribute
<code>aaa_attr_get_value()</code>	Gets the attribute value
<code>aaa_attr_get_value_binary()</code>	Gets the attribute's value in binary
<code>aaa_attr_get_value_string()</code>	Gets the length and contents of the attribute pointed to by the cursor
<code>aaa_attr_get_value_ulong()</code>	Gets the attribute values which are of type <code>ulong</code>
<code>aaa_attr_has_valid_tag()</code>	Checks whether the value passed for the tag is valid
<code>aaa_attr_list_alloc()</code>	Gets the ID manager handle associated with an attribute list
<code>aaa_attr_list_copy()</code>	Invokes a wrapper function for <code>aaa_attr_list_copy_internal()</code> (which copies a list and returns the handler)
<code>aaa_attr_list_copy_from_req()</code>	Copies a list using the ID from the request structure
<code>aaa_attr_list_copy_to_req()</code>	Copies a list to a request structure
<code>aaa_attr_list_free()</code>	Frees the list and make the handler <code>ID_MGR_INVALID_HANDLE</code>
<code>aaa_attr_list_merge()</code>	Uses a client-safe version of <code>aaa_merge_lists()</code>
<code>aaa_attr_list_ptr_copy_from_req()</code>	Gets the ID manager handle associated with the given attribute list
<code>aaa_attr_list_ptr_copy_to_req()</code>	Copies the list to the req structure
<code>aaa_attr_list_replace_for_req()</code>	Invokes a wrapper for <code>aaa_attr_list_replace_for_req_internal()</code>
<code>aaa_attr_list_set_to_req()</code>	Copies the given list to the request even if there is already a list in the request
<code>aaa_attr_mandatory()</code>	Finds out if an attribute is mandatory or not
<code>aaa_attr_peruser_handler()</code>	Adds perUser-specific attributes to the perUser list
<code>aaa_attr_req_add()</code>	Adds the attributes to the req structure

Table 31-1 AAA Attribute Functions (continued)

Function	Description
aaa_attr_req_add_string()	Adds a string type attribute to the req structure
aaa_attr_req_add_ulong()	Adds a ulong type attribute to the req structure
aaa_attr_to_pak()	Concatenates all contiguous attributes that have the same type as the attribute pointed to by the cursor
aaa_cursor_init_from_attr_list()	Gets the list, initiates the cursor, allocates corresponding memory and sets the initial values
aaa_cursor_init_from_req()	Initiates the cursor, allocates memory and sets the service, protocol and ID
aaa_get_abort_cause_string()	Gets the abort cause
aaa_get_terminate_cause_string()	Gets the reason behind termination
aaa_merge_lists()	Appends an attribute list to another attribute list
aaa_tagged_attr_event_add()	Encodes a AAA attribute with a single byte tag for the given event
aaa_tagged_attr_event_add_ulong()	Encodes a AAA attribute with a single byte tag for a given event
aaa_util_attr_replace_without_newinfo()	Updates a given attribute without generating an accounting record given a unique ID and type attribute

31.2.2 AAA Utility Functions

Table 31-2 describes the AAA utility functions

Table 31-2 AAA Utility Functions

Function	Description
aaa_find_hwidb_by_session_id()	Finds the HWIDB corresponding to a given session ID
aaa_is_preath_callback()	Checks if the callback flag in the preauth DB is set or not
aaa_util_acct_method_list_to_index()	Gets the method list from using the method list name
aaa_util_add_apply_config_to_handler()	Adds the apply flag for the peruser handler context
aaa_util_add_authen_db()	Creates an authen DB and add the authen data to the authen DB
aaa_util_add_idb_to_handler()	Adds the IDB to the peruser handler context
aaa_util_add_notify_pid_to_handler()	Adds the PID to the peruser handler context
aaa_util_add_subintf_to_handler()	Adds the subinterface descriptor block to the peruser handler context
aaa_util_alloc_acct_response()	Gets the handle to the allocated aaa_acct_response data
aaa_util_associate_acctresp_event()	Associates an accounting response struct with an event struct

AAA Functions

Table 31-2 AAA Utility Functions (continued)

Function	Description
aaa_util_associate_ctxt_acctresp()	Associates a client-specific context with accounting response struct
aaa_util_authen_index_is_method_present()	Searches an AAA authentication method list for a given method
aaa_util_authen_is_default_mlist_defined()	Determines if there is a “default” list defined for the service
aaa_util_authen_is_mlist_defined()	Determines if there is a list defined for the service of the given name
aaa_util_authen_method_list_to_index()	Gets the method list by using the method list name
aaa_util_author_data_avail_for_protocol()	Copies the author data from the interface DB in the case where authen and author methods are both radius
aaa_util_author_data_available_for_type()	Checks if authorization data is available
aaa_util_author_index_is_method_present()	Searches a AAA authorization method list for a given method
aaa_util_author_is_mlist_defined()	Determines if there is a list defined for the service of the given name
aaa_util_author_merge_preath_author()	Merges preauth author data into authen author data
aaa_util_author_merge_service_profile_author()	Ties a local AAA server service profile to a user authorization list
aaa_util_author_method_list_to_index()	Gets the method list from using the method list name
aaa_util_author_needed()	Checks if authorization is required on a connection
aaa_util_cas_unique_id_alloc()	Allocates a unique ID for CAS
aaa_util_check_acct_action()	Checks whether the indicated action has been set for the call
aaa_util_clear_multi_stage_auth_info()	Clears out any held state used during multi-stage authentication
aaa_util_dealloc_acct_response()	Deallocates the aaa_acct_response data structure
aaa_util_debug_auth_type()	Debugs the contents of an authentication type array encoded by aaa_util_set_auth_type()
aaa_util_delete_attr_by_type()	Deletes an attribute from an accounting type record
aaa_util_delete_multi_attr_from_protocol()	Deletes and free a specified attribute from the protocol database
aaa_util_delete_string_session_id()	Deletes the session ID from an accounting type record
aaa_util_do_preath()	Determines whether the client should make a preauth call or not
aaa_util_free_handler_ctx()	Frees the handler for the peruser
aaa_util_free_preath_author_db()	Frees the preauth DB given a handle
aaa_util_get_acct_session_id_from_list()	Retrieves the account session ID attribute from aaa_attr_list

Table 31-2 AAA Utility Functions (continued)

Function	Description
aaa_util_get_attr_list_from_protocol()	Gets a list containing a specified attribute from the protocols databases
aaa_util_get_attr_name_from_type()	Gets the attribute name from the enumeration attribute type
aaa_util_get_attr_type_from_name()	Gets the enumeration attribute type from the attribute name string
aaa_util_get_authen_method()	Returns the authentication method used to derive the data associated with the AAA unique ID
aaa_util_get_clid_dnis()	Gets the CLID and DNIS values from general_db
aaa_util_get_cmdlist()	Gets a new command list
aaa_util_get_component_type()	Gets the component type of the owner of a unique ID
aaa_util_get_conn_progress()	Returns the last connection progress set by the client for this session
aaa_util_get_disconnect_cause()	Retrieves the very first disconnect cause code set by anyone tearing down this call
aaa_util_get_hwidb()	Gets the hardware IDB
aaa_util_get_orig_component_type()	Gets the component that currently owns the unique ID
aaa_util_get_peer_ip_addr()	Finds the peer IP address in the accounting database
aaa_util_get_preath_attr_by_type()	Gets preauth attribute names
aaa_util_get_preath_attr_ulong_by_type()	Extracts integer-value AAA attributes from the AAA preauth database authorization lists
aaa_util_get_preath_authen_authtype()	Gets the authtype from the preauth data in the AAA DB
aaa_util_get_preath_author_data()	Gets the attribute list handle for the authorization data from preauthentication
aaa_util_get_preath_author_db()	Gets a handle for a preauth DB copy associated with a AAA unique ID
aaa_util_get_service_profile_list()	Gets the authorization list found in the service profile DB
aaa_util_get_session_id()	Gets the session ID given the AAA unique ID and the account type
aaa_util_get_stored_session_id()	Retrieves the account session ID from AAA given the aaa_acct_type and aaa_unique_id
aaa_util_get_tableid()	Gets the routing table ID from aaa_unique_id in per-VRF AAA
aaa_util_get_tty_from_uid()	Returns the pointer to the TTY struct given an aaa_unique_id
aaa_util_get_unique_id()	Adds the accounting session ID for a client
aaa_util_if_needed_configured()	Checks if a client needs to do one more authentication
aaa_util_init_handler_ctx()	Initiates the per-user context
aaa_util_interface_disconnect()	Disconnects different types of interfaces

AAA Functions

Table 31-2 AAA Utility Functions (continued)

Function	Description
aaa_util_is_seperate_author_needed()	Searches a AAA authorization method list to determine if separate authorization requests are required for each NCP
aaa_util_is_session_available()	Finds whether a given session is associated with the given AAA UID
aaa_util_is_user_exec_authenticated()	Determines whether the user is exec authenticated
aaa_util_override_attr()	Overrides the UID DB attributes with the new values specified
aaa_util_override_session_id()	Generates a copy of the name override session ID
aaa_util_pick_and_save_class_from_list()	Retrieves the class attribute for the callback call
aaa_util_preath_authen_needed()	Checks if we need to do authentication after preauth has been done
aaa_util_preath_author_data_exists()	Indicates if any kind of preauthorization info exists
aaa_util_preath_multilink_negotiate()	Checks if preauthentication requires multilink to be enabled
aaa_util_preath_verify_service_type()	Checks whether authentication is needed or not
aaa_util_promting_needed()	Validates the cursor and check if the attribute PROMPT flag is set
aaa_util_put_preath_author_db()	Associates a preauth DB with the given AAA ID
aaa_util_remove_id_from_idb()	Removes the ID from the IDB
aaa_util_return_acct_status()	Gets the account status
aaa_util_set_auth_type()	Sets the authentication type
aaa_util_set_clid()	Sets the CLID value in the AAA general DB
aaa_util_set_conn_progress()	Sets the connection progress value for this session
aaa_util_set_disconnect_cause()	Sets the appropriate disconnect cause
aaa_util_set_event_timestamp_local_timezone()	Sets the flag aaa_event_timestamp_local_timezone which indicates whether we need to do local timezone adjustment
aaa_util_store_session_id_extension()	Passes prefix or suffix string to be added to the account session ID
aaa_util_tty_changed()	Moves a TTY call from TTY to another
aaa_util_unique_id_alloc()	Allocates a unique AAA ID and associate it with a hwidb and/or a TTY
aaa_util_unique_id_dealloc()	Deallocates the unique ID allocated after the stop of a session
aaa_util_update_mlp_bundle()	Updates the multiple PPP session bundle

31.2.3 Authentication and Authorization Functions

Table 31-3 describes the authentication and authorization functions.

Table 31-3 Authentication and Authorization Functions

Function	Description
aaa_abort_login_authen()	Performs a login authentication after an abort
aaa_event_acct_init()	Sets the event type for an accounting request
aaa_event_alloc()	Gets a AAA event handle
aaa_event_free()	Frees an event
aaa_event_init_abort_req()	Cancels a previous authorization request
aaa_event_send_and_free()	Posts an accounting event or free the event if AAA_EVENT_FLAG_DONT_SEND is set
aaa_event_set_passthru()	Sets the event as a passthru type
aaa_get_req_status_detail()	Gets a request's status in detail
aaa_req_alloc()	Builds a AAA request
aaa_req_authen_init()	Initializes the authentication
aaa_req_author_init()	Sets the attributes for authorization
aaa_req_free()	Frees all local data related to a request
aaa_req_get_authen_method()	Gets the current authen method rather than the authen list
aaa_req_get_callback()	Gets a callback function for a request
aaa_req_get_context()	Gets the context attribute from the structure
aaa_req_get_list()	Gets the list handle
aaa_req_get_protocol()	Gets the protocol type
aaa_req_get_status()	Gets the request status
aaa_req_handle_get_author_mlist()	Gets the authorization method list handle
aaa_req_process_conn_author()	Processes the authorization data, if any, for reverse telnet authorization
aaa_req_send()	Sends a request to AAA
aaa_req_send_blocked()	Blocks the lists which are currently not of interest
aaa_req_send_spoofed_reply()	Spoofs a AAA reply message
aaa_req_set_authen_action()	Sets the structure with authentication action
aaa_req_set_authen_service()	Sets the required authentication service type
aaa_req_set_callback()	Sets the callback function
aaa_req_set_context()	Sets the context attribute in the structure
aaa_req_set_force_chal_attr()	Forces AAA to send the CHAP Challenge as an attribute instead of sending it as an authenticator
aaa_req_set_passthru()	Sets the request as a passthru type

31.2.4 AAA Profile Functions

Table 31-4 describes the AAA profile functions.

AAA Functions**Table 31-4 AAA Profile Functions**

Function	Description
aaa_filter_handle_create()	Returns a filter handle
aaa_filter_handle_free()	Frees the filter handle
aaa_filter_set()	Turns blocking of an attribute on or off
aaa_profile_handle_create()	Creates a profile handle
aaa_profile_set_for_event()	Attaches a profile to a AAA event
aaa_profile_set_universal_output_filter()	Sets the specified filter as a universal output attribute filter for the specified profile

31.2.5 Server Group Functions

Table 31-5 describes the server group functions.

Table 31-5 Server Group Functions

Function	Description
aaa_server_v2_create_handle()	Creates a private server handle
aaa_sg_v2_add_server()	Adds a server to a server group
aaa_sg_v2_all_servers_are_dead_and_need_msg()	Determines if an ALLDEADSERVER message should be produced
aaa_sg_v2_count_servers()	Counts the number of servers in a server group
aaa_sg_v2_create_private()	Creates a handle for a private (nvgen'd) server group
aaa_sg_v2_create_public()	Creates a handle for a public (nvgen'd) AAA server group
aaa_sg_v2_do_alias_lookup()	Determines whether to perform alias lookups for servers in the group
aaa_sg_v2_find_server_in_group()	Finds a particular server in a server group by specifying the address and ports of the server
aaa_sg_v2_get_deadtime()	Gets the length time for which the server will be dead
aaa_sg_v2_get_name()	Returns the name of a server group
aaa_sg_v2_get_nas_port()	Gets the NAS port type to be sent by this server group
aaa_sg_v2_get_nas_port_string_format()	Gets the NAS port type string format to be used by this server group
aaa_sg_v2_get_server_type_string()	Produces a string describing the type of server in a group
aaa_sg_v2_get_source_idb()	Gets the NAS port type to be used by this server group
aaa_sg_v2_get_type()	Determines the type of server which a server group holds
aaa_sg_v2_group_handles_are_equal()	Compares two server group handles

Table 31-5 Server Group Functions (continued)

Function	Description
aaa_sg_v2_name_and_type_str()	Gets the name and type of a server group used for debug output and nvgen
aaa_sg_v2_need_to_print_live_message()	Decides whether to print a message indicating that not all servers in the group are dead
aaa_sg_v2_public_lookup()	Looks up the handle for a public (nvgen'd) AAA server group
aaa_sg_v2_remove_server()	Removes a server from a server group
aaa_sg_v2_server_add_alias()	Adds an IP alias to a server
aaa_sg_v2_server_get_address()	Determines the address of the server
aaa_sg_v2_server_get_timeout()	Determines how long to wait for a response from the server
aaa_sg_v2_server_handle_free()	Indicates that a server handle will no longer be used
aaa_sg_v2_server_handles_are_equal()	Compares two server handles
aaa_sg_v2_server_set_acct_port()	Sets the port to which accounting information should be set
aaa_sg_v2_server_set_address()	Sets the address associated with a server handle
aaa_sg_v2_server_set_authen_port()	Sets the authentication port in the server
aaa_sg_v2_server_set_author_port()	Sets the authorization port in the server
aaa_sg_v2_server_set_key()	Sets the key used for communication with the server
aaa_sg_v2_server_set_type()	Sets the type of server
aaa_sg_v2_set_alias_lookup()	Specifies whether to do DNIS alias lookup
aaa_sg_v2_set_deadtime()	Sets the per-group dead time for the server
aaa_sg_v2_set_max_tries_per_transaction()	Sets the maximum number of attempts that a given transaction will make to reach a server
aaa_sg_v2_set_nas_port()	Sets NAS port type to be sent by this server group
aaa_sg_v2_set_nas_port_string_format()	Sets the NAS port type string format to be used by this server group
aaa_sg_v2_set_retransmit_interval()	Sets the default retransmit interval for transactions against this server group
aaa_sg_v2_set_server_nonstandard()	Indicates that a server should use nonstandard (ascend-style) attributes
aaa_sg_v2_set_server_pick_order()	Sets the order in which servers are chosen
aaa_sg_v2_set_server_retransmit_interval()	Sets the amount of time that AAA should wait for a server to reply to a request
aaa_sg_v2_set_source_idb()	Sets a source interface (i.e., address) with the server group
aaa_sg_v2_set_tries_per_server()	Sets the number of attempts which will be made to contact a server in a row before failing over to the next server
aaa_sg_v2_set_vrf_by_name()	Sets the VRF associated with a server group by name

Authentication Call Flow

Table 31-5 Server Group Functions (continued)

Function	Description
aaa_sg_v2_set_vrf_by_tableid()	Sets the VRF associated with a server group using the VRF table ID
aaa_sg_v2_sg_handle_free()	Indicates that a server group is no longer needed

31.2.6 AAA Accounting Functions

Table 31-6 describes the AAA accounting functions.

Table 31-6 AAA Accounting Functions

Function	Description
aaa_acct_abort_type()	Cleans up accounting information
aaa_acct_attr_delete()	Removes the specified accounting attribute from the AAA database
aaa_acct_create_req()	Creates an accounting request
aaa_acct_ipsec_tnl_action()	Handles ipsec-tunnel accounting action events
aaa_acct_map_vpdn_action_to_status()	Maps a AAA action to the accounting status type
aaa_acct_post_event_vpdn_tunnel_or_link()	Posts VPDN tunnel or link accounting events
aaa_acct_send_passthru_req()	Sends an accounting event in the case that radius passthru is specified
aaa_acct_set_mlist()	Sets the method list for accounting
aaa_acct_set_mlist_passthru()	Sets the method list for a passthru event
aaa_acct_set_periodic_interval()	Sets a period interval for accounting updates
aaa_send_all_stop_requests()	Sends a stop event for each accounting record within the given account DB except command accounting
aaa_util_replace_acct_session_id()	Replaces the session ID
multi_session_free_rec()	Clears out all links of a multi session

31.3 Authentication Call Flow

```
#include <stdio.h>

int aaa_authen_flow() {
    /* This function describes the authentication call flow. */
    /* Build, send and free the request. */
    aaa_id = aaa_util_unique_id_alloc (IDB, TTY, aaa_component_test_flow,
        pointer_to_cli_handle);
    req = aaa_req_alloc (PID);
    authen_list = aaa_util_authen_method_list_to_index (name,
        aaa_authen_type);
    ret = aaa_req_authen_init (req, service_type, action_type,
        aaa_authen_type, aaa_id, authen_list, author_list, acct_list);
    aaa_req_set_context (req, handle_to_context);
```

```

aaa_req_set_callback (req, aaa_receive_authen_response);
aaa_cursor_init_from_req (pointer_to_attr, req, aaa_service_type,
    aaa_protocol_type);
ret = aaa_attr_get_by_type (pointer_to_attr, attr_type);
ret = aaa_attr_req_add_ulong (req, attr_type, value, attr_option_flag);
ret = aaa_attr_req_add_string (req, attr_type, user, strlen(user),
    attr_option_flag);
ret = aaa_req_send (req);
handle = (handle_type) aaa_req_get_context (req);
status = aaa_req_get_status (req);
aaa_attr_list_free (list, aaa_id);
list = aaa_attr_list_copy_from_req (req);
aaa_util_author_merge_preath_author (aaa_id, author_data_list);
aaa_util_set_conn_progress (aaa_id, AAA_AV_CONN_PROG_);
aaa_req_free (req);
}

/* This function describes the authentication response callback */
int aaa_receive_authen () {

    /* This puts the process to sleep until an asynchronous event awakens it. */

    while (TRUE) {
        process_wait_for_event ();

        /* process_get_wakeup finds out the next reason why a process was awakened */

        while (process_get_wakeup (class, queue)) {

            switch (class) {
                case message_event:

                    /* Processes all pending messages */
                    while (process_get_message (message, pointer_to_arg,
                        value)) {

                        /* It gets the next message outstanding for this process */

                        switch (message) {
                            case aaa_reply_message:

                                /* This gets the context attribute from the structure */

                                handle = (handle_type) aaa_req_get_context (req);

                                /* It requests the execution of an event handler using the context
                                information provided */

                                if (req) {
                                    ret = evt_request (event, CLIENT_AAA_RESPONSE, 0,
                                        0, handle_to_context, pointer_data);
                                }
                                break;
                            }
                            break;
                        }
                    }
                break;
            }
        }
    }
}

/* Assuming the event has been already registered as ... */

```

```

/* This registers the event type */

(void) evt_register (client_aaa_response, client_aaa_message_eh,
                     client_handle_name, FALSE, client_event_queue, "AAA Response");

/* evt_elem is the queue element that is common to all events/queues in the
Event Manager system */

static void client_aaa_message (const evt_elem event)
{
    void (pointer_to_callback)(aaa_req_handle req);
    aaa_req_handle req;

    req = (aaa_req_handle) event_data;

    /* This gets the callback function */

    callback = aaa_req_get_callback (req);

    if (callback) {
        (pointer_to_callback) (req);
    }
}

```

31.4 Authorization Call Flow

The authorization call flow is very similar to that of the authentication call flow. All the APIs called are the same as authentication except for a few APIs.

To get the authorization method list, **aaa_util_author_method_list_to_index()** is called wherein the syntax of which is the same as its authentication equivalent. And secondly, to initialize authorization, **aaa_req_author_init()** is called. Otherwise, it shares the same call flow as authentication.

31.5 Accounting Call Flow

```

#include <stdio.h>

/* This function describes the accounting call flow of AAA APIs */
int aaa_acct_flow( ) {

    /* This allocates an unique id to the session for tracking */
    aaa_id = aaa_util_unique_id_alloc (IDB, TTY, aaa_component_test_flow,
                                       pointer_to_cli_handle);

    /* This gets the method list for accounting from the method list name */
    acct_list = aaa_util_acct_method_list_to_index (name, aaa_acct_type);

    /* This stores the caller PC for use in any malloc*( )s that AAA performs
whilst in the context of that API call. */
    event = aaa_event_alloc ( );

    /* This function initializes sending accounting requests to AAA. */

```

```
status = aaa_event_acct_init (event, action, aaa_acct_type, aaa_id);

/* The following two functions add the respective type values to the event */

ret = aaa_attr_event_add_ulong (event, attr_type, value);
ret = aaa_attr_event_add_string (event, attr_type, username,
strlen(username));

/* This function creates a req */

req = aaa_acct_create_req (method_list, aaa_acct_req_flag);

/* This function initiates the abort request setting up the corresponding
values */

ret = (void) aaa_event_init_abort_req (event, req);

/* If not in passthru then all memories and lists are freed, else a passthru
event is sent out immediately without adding additional radius attributes */

ret = aaa_event_send_and_free (event);
if (passthru) {

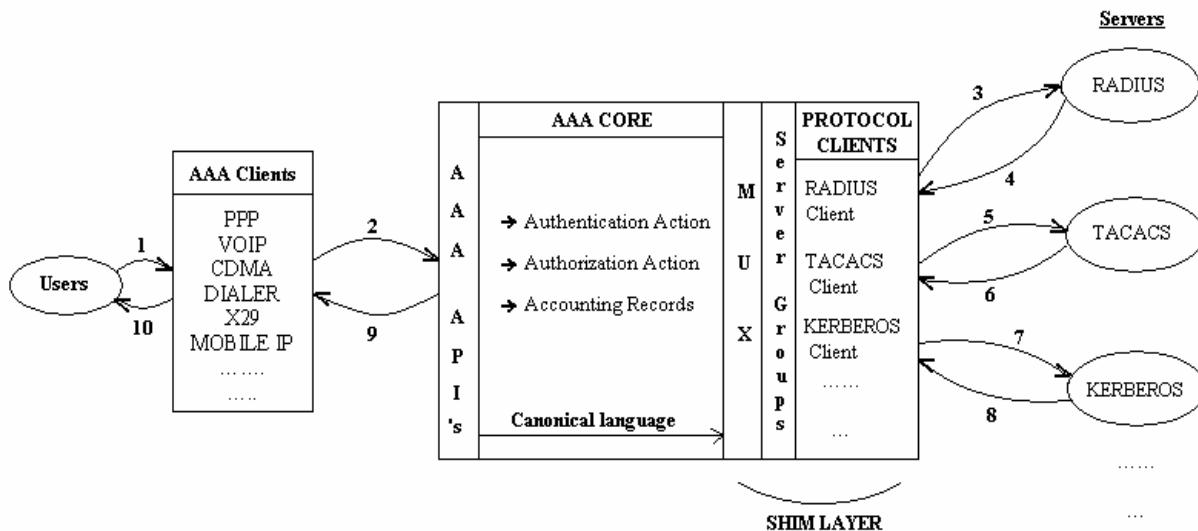
/* This function is used to send an accounting event in the case that radius
passthru is specified */
ret = aaa_acct_send_passthru_req (event);
}
}
```

31.6 AAA Architecture Diagram

Figure 31-1 shows the overall architecture of AAA.

Process Interaction Diagram

Figure 31-1 AAA Architecture

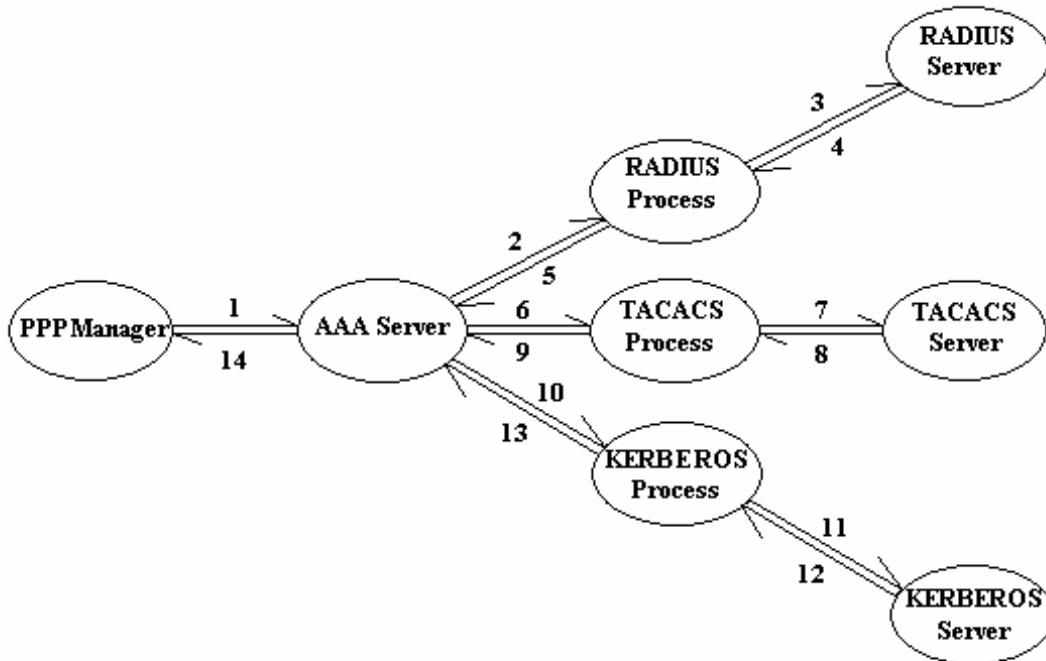


- 1 Users get authenticated for the required access through one of the various AAA clients as indicated in the above diagram.
- 2 The AAA clients communicate to the AAA core by calling the AAA APIs which carry out the functions for authentication. The AAA core processes the request from the AAA clients through a canonical language and converts and assigns the packet to the respective server group through a MUX.
- 3, 5 and 7 The protocol clients at the shim layer of the AAA server communicate to their respective servers for the required AAA process.
- 4, 6 and 8 The servers communicate back to the AAA server through the call back function where the response path is the exact reverse of the request path.
- 9 The AAA server converts the response from the respective server and asks the AAA client for the required information, if any.
- 10 The client in turn forwards the challenge to the users when necessary.

31.7 Process Interaction Diagram

Figure 31-2 shows the process interactions in AAA using PPP as the use case AAA client.

Figure 31-2 Process Interaction

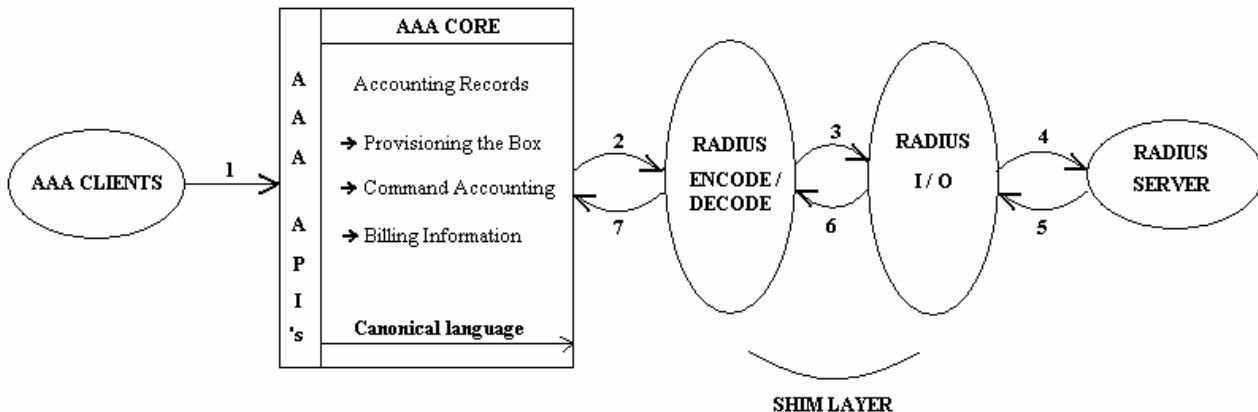


- 1 The PPP manager contacts the AAA server for authentication.
- 2, 6 and 10 The AAA server instigates its APIs to perform the required function and submit the request message to the appropriate server process. RADIUS uses UDP whereas TACACS uses TCP. So depending upon the process and function required, the appropriate server is chosen.
- 3, 7 and 11 The initiated server process now converts the request into its own respective server format and then communicates with the server.
- 4, 8 and 12 The server replies back to the respective server process regarding the pending request.
- 5, 9 and 13 The server process converts the packet back to the AAA format and forwards it to the AAA server.
- 14 The AAA server in turn forwards the response from the server to the PPP manager which is the AAA client here.

31.8 AAA Accounting Model Using RADIUS

Figure 31-3 shows an example of AAA accounting using RADIUS.

Figure 31-3 AAA Accounting Model Using RADIUS



- 1 AAA clients post events to the AAA core using AAA APIs wherein the events can be SERVICE_UP and SERVICE_DOWN and so on. Along with these events, AAA clients can add the attributes that need to be passed on to AAA.
- 2 The AAA core analyzes the validity of the accounting request, that is, whether this event should be translated into an accounting packet or not. If yes, it sends a request to the RADIUS encode/decode routing with a list of attributes in the AAA format. This list of attributes will contain the attributes added by AAA clients and those added by the AAA core.
- 3 RADIUS encode/decode routing will translate those attributes from AAA format to RADIUS format and add RADIUS specific attributes to the req. Then, it throws the req into the inQ from which the RADIUS I/O routine picks up the req.
- 4 RADIUS I/O routine does an `ip_udp_send()` and also adds this entry to the wait ctx which is hashed on (IP address of RADIUS server, source port of the req, rad ident) and starts the timer corresponding to the radius server for timeouts. This wait ctx is used to check if there is a valid request that exists for an incoming response from the RADIUS server. If timeouts happen and if the RADIUS server is changed for failover, this module moves the wait ctx from one hash bucket to the other.
- 5 The RADIUS server acknowledges the accounting request and sends back a response to the RADIUS I/O.
- 6 and 7 The incoming RADIUS response is translated into the AAA format and sent to the AAA core.

31.9 Basic AAA Configurations

At the NAS configuration menu, a new model is instigated which enables new access control commands and functions:

```
NAS(config)# aaa new-model
```

These commands set up RADIUS servers by specifying their IP addresses, their respective UDP ports for authentication and accounting and the per-server encryption keys:

```
NAS(config)# radius-server host 2.1.1.4 auth-port 1062 acct-port 1061 key 1234
NAS(config)# radius-server host 2.1.1.5 auth-port 1001 acct-port 1022 key 2345
```

This defines and sets a RADIUS server group `sample_sg`:

```
NAS(config)# aaa group server radius sample_sg
```

These add the respective servers to the specified server group.

```
NAS(config-sg-radius)# server 2.1.1.4  
NAS(config-sg-radius)# server 2.1.1.5
```

This specifies the dead time of the server group in minutes.

```
NAS(config-sg-radius)# deadline 20
```

This sets the method list for authentication by associating it with the required server groups.

```
NAS(config)# aaa authentication ppp authen_mlist group sample_sg
```

This sets the method list for authorization by associating it with the required server groups, to instigate network services for authorization of the PPP session.

```
NAS(config)# aaa authorization network author_mlist group sample_sg
```

This sets the method list for accounting by associating it with the required server groups, to instigate network services for start-stop accounting for the PPP session.

```
NAS(config)# aaa accounting network acct_mlist start-stop group sample_sg
```

31.9.1 PPP Configuration between Client and NAS Using PAP and CHAP

At the client's side:

```
Router(config)# hostname Client  
Client(config)# interface Serial2/0  
Client(config-if)# shutdown
```

Sets the IP address of the serial interface wherein the subnet mask is also specified.

```
Client(config-if)# ip address 171.10.1.2 255.255.0.0
```

Sets the encapsulation type for the serial interface.

```
Client(config-if)# encapsulation ppp
```

```
if (PAP) {
```

Sets the sent username and password for PAP.

```
Client(config-if)# ppp pap sent-username client password bangalore4
```

```
}
```

```
if (CHAP) {
```

Sets the PPP authentication type to CHAP.

```
Client(config-if)# ppp authentication chap
```

Basic AAA Configurations

Establishes the username and password authentication.

```
Client(config)# username NAS password bangalore4
}
}
```

Brings up the interface and hence the PPP session.

```
Client(config-if)# no shut
```

At the NAS's side:

```
Router(config)# hostname NAS
NAS(config)# interface Serial3/0
NAS(config-if)# shutdown
NAS(config-if)# ip address 171.10.1.1 255.255.0.0
NAS(config-if)# encapsulation ppp

if (PAP) {
    /* Sets the PPP authentication type to PAP */

NAS(config-if)# ppp authentication pap

}
if (CHAP) {

NAS(config-if)# ppp authentication chap

/* Establishes the username and password authentication */

NAS(config)# username client password bangalore4
}
NAS(config)# radius-server host 2.1.1.4 auth-port 1062 acct-port 1061 key 1234
NAS(config)# radius-server host 2.1.1.5 auth-port 1001 acct-port 1022 key 2345
NAS(config)# aaa group server radius sample_sg
NAS(config-sg-radius)# server 2.1.1.4
NAS(config-sg-radius)# server 2.1.1.5
NAS(config-sg-radius)# deadtime 20
NAS(config)# aaa authentication ppp authen_mlist group sample_sg
NAS(config)# aaa authorization network author_mlist group sample_sg
NAS(config)# aaa accounting network acct_mlist start-stop group sample_sg
NAS(config-if)# no shut
```

P A R T 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.

Note Cisco IOS scalable process questions can be directed to the ios-scalers@cisco.com and scalers@cisco.com aliases.

32.1 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.

32.2 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

The Typical Scenario

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.

32.2.1 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

32.2.1.1 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.

32.2.1.2 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.

32.2.1.3 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 awakened at 4 a.m. Although fascinating in the same morbid sense as a multicar pileup, this scenario is one to be avoided.

32.2.1.4 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.

32.2.1.5 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.

32.2.1.6 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.

32.3 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.

32.3.1 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 milliseconds 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 enqueueing 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.

32.3.2 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.

32.3.3 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.

32.3.4 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.

32.3.5 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.

32.4 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 (EHS), which is a redundant processor scheme, in which one processor takes over when the other dies or crashes. See “Enhanced High System Availability (EHS)” in Chapter 2, “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.

33.1 Overview

This section describes how the backup system operates and how you configure a backup interface.

33.1.1 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 *secondary*. 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.

33.1.2 Configuring Interfaces

This section details how the interfaces are configured:

- Specifying the Standby Interface
- Specifying Backup Delays
- Specifying Backup Loads, Main Interfaces Only

33.1.3 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.

33.1.4 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 never`. 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 delay**.

33.1.5 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.

33.1.6 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.

Description of Changes

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.

33.2 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 parameters 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.

Description of Changes

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.

Verifying Cisco IOS Modular Images

This chapter is no longer relevant and is now deprecated. (June 2007)

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 to verify the Cisco IOS source code modularity using modular image targets in the `sys/makefile` files and in the platform-specific makefiles.

Note Cisco IOS modular image questions can be directed to the ios-images-ready-for-pilot@cisco.com alias.

34.1 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 is 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).

34.2 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

Types of Modularity Checks

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.

34.3 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.

34.4 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.

34.5 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.

34.5.1 Build All Modular Images for a Single Platform

To build all the modular images for a single hardware platform, follow these steps:

Step 1 Change into the platform directory:

```
cd sys/obj-processor-platform
```

Step 2 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 Step 2 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.

34.5.2 Build a Specific Modular Image for a Single Platform

To build a specific modular image for a single hardware platform, follow these steps:

Step 1 Change into the platform directory:

```
cd obj-processor-platform
```

Step 2 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 Step 2 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.

34.6 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 1 Change into the sys directory:

Check Modularity with the sys/scripts/connect Script

```
cd sys
```

Step 2 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 Step 2 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.

34.7 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.

34.8 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`.

Modularity Checking Done by the Nightly Builds

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.

Note Cisco IOS Release-note enclosure questions can be directed to the write-rnes@cisco.com alias.

35.1 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.

35.2 How Customers See Release-note Enclosures

Customers see Release-note enclosures in one of the following ways:

- Cisco Connection Online (CCO). The CCO Bug Navigator, which is part of the Bug ToolKit, 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).

Who Writes Release-note Enclosures

- Cisco Connection Documentation (CCD, formerly UniverCD). This is a monthly CD-ROM produced by the customer documentation groups. For each currently supported Cisco IOS software release, CCD contains a file that lists all the Release-note enclosures.
- Software release notes. These contain 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.

35.3 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.

35.4 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.

35.5 Writing Release-note Enclosures

This section discusses the issues involved in writing Release-note enclosures.

35.5.1 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.

35.5.2 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.

35.5.2.1 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 12.0(2), ...

35.5.2.2 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.

35.5.2.3 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.

35.5.3 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 server*. Do not use abbreviations, such as *c7000* or *7000 series*, or internal code names, such as *Volcano*.
- Refer to specific releases of the Cisco IOS 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.

35.5.4 Text Formatting Guidelines

35.5.4.1 Character Formatting Guidelines

Some scripts gather the Release-note enclosure text for inclusion in formatted documents, such as FrameMaker documents. To properly 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 35-1 explains the character formatting strings. Note that these strings are *not* case-sensitive.

Table 35-1 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>	<i>dialer-string</i>
<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.

35.5.4.2 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 be displayed in error messages.

35.5.5 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.

35.5.6 Guidelines for Using \$\$PREFCS in Release-note Enclosures

Product teams who do not use the CSC.labtrunk/CSC.sys DDTS model may use the \$\$PREFCS. This new option may be used as the first line in a Release-note enclosure to identify those bugs which are found prior to FCS. Any bug with \$\$PREFCS in the Release-note enclosure will be known as a defect found prior to FCS and therefore will not be counted against the BU Release-note enclosure metrics. At FCS, all open bugs with \$\$PREFCS in the Release-note enclosure should be reviewed. Any S1-S3 customer impacting bugs should replace ##PREFCS with a release note. Non-impacting customer bugs should replace ##PREFCS with ##IGNORE. Furthermore, customers should never encounter a bug with ##PREFCS in the Release-note enclosure.

35.5.7 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 as IP, 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-CXBUSER: Slot 12, CBus Error

%RSP-3-RESTART: cbus complex

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

PROBLEMS: 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 sdlc (rsdlc) 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 unbeknownst. And no code change needs to be applied. *engineer's name and date*

PROBLEMS: 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

PROBLEMS: 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.

35.6 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.

35.7 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. See also the ENG-11720 document that is used by engineers as a guide to writing Release-notes.

If you are working with writers from customer documentation, ask if they could help you. If none of these options work, send email to ios-doc.

For additional information about writing Release-note enclosures, see
<http://wwwin-swtools.cisco.com/swtools/ddts>.

For information about using DDTs, see the *Cisco Engineering Tools Guide*
<http://wwwin-enged.cisco.com/common/doc/tools/>.

To enroll in the DDTs course, go to the “Cisco Engineering Training” Web page:
http://wwwin-enged.cisco.com/courses/catalogp/ddts_frame.htm

Small Feature Commit Procedure

This chapter is v3.0.7 of ENG-11517, the Featurette Web page. Refer to the Featurette Web page for the Featurette FAQ and the current lists of known featurettes in a given release of the IOS software:
<http://wwwin-release.cisco.com/relops/Featurette>

36.1 Introduction

This chapter provides guidelines for committing very small features (“featurettes”) into IOS Early Deployment releases. It is not intended to supersede GEM, GEM-Lite, IOS Software Engineering Process, or any other development process document.

36.1.1 Purpose

Cisco’s IOS release processes must be sufficiently flexible to scale appropriately for the size and complexity of software commits. This document provides a streamlined approach to committing “featurettes.”

Tracking features:

- raises visibility to affected organizations such as Dev-Test, Documentation, Customer Advocacy, Marketing, and other BUS
- ensures that customers will be informed about the featurette via customer documentation, which is the only way customers have of getting this information.
- allows Cisco to investigate changes to IOS to avoid lawsuits from other companies who claimed that they invented technology first.
- alerts test teams to plan for the integration of new automated scripts to test featurettes.

36.1.2 Definition

Featurettes are small additions of functionality. As a project’s functionality begins to grow in size, the following guidelines can assist in determining whether or not the functionality is a feature or a featurette.

- Featurettes must:
 - Be less than 2000 lines of new/changed source code. Note: A featurette cannot be broken into smaller featurettes so that each is below the minimum code limit.
 - Be recorded in only 1 per DDTs. Do not put multiple featurettes in one DDTs.

Key Requirements

- Impact only a few subsystems (normally only one). Changes to subsystems other than those owned by the DE manager are to be reviewed by the DE managers who own the other subsystems.
- Be parented and tested on the branch into which they will commit.
- Have a partial Commit Requirements Form if:
 - a feature module or a rewrite of a feature module is required
 - the change impacts any test scripts or causes new scripts to be written.
- Only be committed to a central PI branch or a BU PI branch.
- Featurelettes cannot:
 - Introduce new subsystems or images and cannot remove them.
 - Require significant cross-functional involvement. Featurelettes may not require significant development in more than one subsystem or by more than one BU or require testing by more than one Dev test organization. This would not preclude requiring a small code change in another subsystem that is code reviewed by the responsible parties, but would preclude committing engineering resources in another BU to co-develop.
 - Be committed directly into a Technology train.
- Featurelettes can:
 - Modify form and function, including user interface changes.
 - Require an EFT
 - Require automated test scripts to be written or updated.

36.2 Key Requirements

The following steps are required whenever featurelettes are committed to Cisco IOS releases:

- The size and complexity of featurelettes are to be limited to the featurette definition at the beginning and the end of development prior to commit.
- A featurette impacting documentation or test scripts requires a partial Commit Requirements Form. A featurette will *NOT* have a feature module written for it nor will major rewrites of existing documentation occur UNLESS there is a partial Commit Requirements Form completed. Documentation will schedule according only to this form.
- The featurette must be tested against a recent label that is within two weeks of the planned commit date to ensure that the featurette will not conflict with the codebase at the time of the commit.
- Featurelettes must have a plan for automating the testing of the featurette or updating existing automated regression scripts. ARF uses the Commit Requirements Form database to track Cisco's test coverage. It is imperative to continue to test featurelettes to improve quality.
- All Severity defects that are caused by the new featurette must be resolved and retested. No new defects can be outstanding at the time of the code commit.
- One week before the commit, the accountable DE manager must e-mail a pointer to the diffs to the ED Train mailing alias (e.g. del-info) so that others have a chance to comment on the new functionality.

- The accountable DE manager is responsible for deciding if new functionality qualifies as featurette (vs. a feature) and for ensuring compliance with the policy.
- The commit review process for featurettes is the responsibility of the DE manager. All featurette meetings and communication to the marketing, test, documentation, and the development community is the responsibility of the DE manager.
- Featurettes represent new functionality and, as such, are to be committed only in Early Deployment (ED) releases, such as 12.1T. No new functionality is to be committed to major (mainline) releases.
- Featurettes must be adequately documented, via the use of appropriate DDTs enclosures (as defined below), to provide needed cross-functional communication of key information. See the required DDTs enclosure section below.

36.3 Commit Requirements Form for Featurettes

If a featurette's functionality is to be marketed, requires a feature module to be written, a rewrite of a feature module, a new test script, or a rewrite of a test script, then a partial Commit Requirements form is required. The following questions are required to be answered.

- 1.1) Feature Name: "Featurette:" <project name>
- 1.1.1) Marketing Feature Name: <EXACTLY what will be advertised to customers>
- 1.3) Summary Description: Enter the DDTs # :
- 1.7) Requested Release: e.g. 12.1-T Train
- 1.8) Requested Maintenance Commit Window : e.g. (03)
- 1.9) Supported Platform(s)
- 1.12) Technology Area: from drop down list.
- 1.13) Development Manager
- 1.14) Test Manager
- 1.17) Product Marketing Manager
- 1.18) Feature Module Writer
- 1.19) IOS Doc Technology Owner

3.6.2) Specify the necessary and sufficient defining subsystems required for this feature to be present in an image. <put the exact .o filenames! This is most important question on the whole form. This is how the functionality is linked to an image in the database)

Future Questions: Who is responsible for creating the automated test regression script? What testbed will it run it? When will the test script be completed.

These forms will be used to drive tracking of automated scripts, generating documentation, and updating the feature database.

36.4 Featurette Commit Window

Featurettes should be committed into a pre-integration branch during the BU's commit window. By committing their features and featurettes together, the BU can test the code to ensure that there will not be any conflicts between the features and featurettes. If the featurette cannot be committed into

a BU pre-integration branch, then it must be committed to a central pre-integration branch. Featurettes are *not* allowed to be committed directly into the shipping T train (such as del_t), since this introduces risk without much time to recover from problems.

36.4.1 DDTs Required Fields and Enclosures

The following DDTs fields and enclosures are required for any featurette.

The following guidelines for DDTs usage should be followed for featurette commits.

- **Project.** Set the project to CSC.labtrunk. Both CSC.labtrunk and CSC.sys support the commit type field to indicate that the DDTs is a featurette.
- **Headline.** Should be written to enable easy identification and understanding by cross-functional groups.
- **Commit Type Field.** Enter the word “featurette,” not fix, feature, or technology.
- **To Be Fixed Field (Must Include Version Number).** The to-be-fixed field must have the correct maintenance release and version. For example, if there is a featurette for 12.1(2)T, then the “to-be-fixed” field should say 12.1(2)T. An entry of 12.1T would be insufficient.
- **Code Reviewer Field.** The featurette must be code reviewed and the names of those giving approval must be documented. Entering an alias in the code reviewer field is insufficient.
- Description Enclosure. A thorough description of the background, function, and key characteristics of the featurette. If desired, the Functional-Spec can be incorporated into the description enclosure.
- **Documentation Enclosure.** A featurette requires a documentation enclosure in the DDTs if documentation is needed.

The documentation enclosure will indicate:

- Hardware platforms supported
- EDCS numbers of related documentation

Furthermore, if a featurette introduces or changes a command or command output, the following information must be provided for each new or changed Cisco IOS command in a “Documentation” enclosure:

1. Command syntax of all new and modified commands
2. What the command does
3. Under what conditions the command is used
4. When it should NOT be used
5. A configuration example; an example of the command
6. If a “show” command is modified, a table must be provided which describes each field.

- **Functional-Spec Enclosure.** The DDTs for each featurette should contain a complete record of the user interface changes and additions in a Functional Specification enclosure. This enclosure should be done 3 weeks before resolution for a normal featurette and 6 weeks before resolution for a marketable featurette.
- **Test Plan Enclosure.** A test plan or pointer to the CMS test plan doc approved by Dev/Test including in the enclosure;
- **Test Results Enclosure.** A summarization of the test results which confirms the quality of the code; could also include the actual test results or a pointer to where the results can be found.

- **Interest List.** The DDTs “interest” field include those functions impact potentially impacted by the featurette commit, such as: Customer Assurance, Cross-Sector Documentation Team alias (iosdoc-xsector-team), Marketing/Sales, Training/Support, Test, DE code reviewers, etc.

36.5 Featurette Mailing Alias

Anytime a new featurette DDTs is filed or when a featurette DDTs changes state, email is issued to the featurette-interest alias.

36.5.1 Email Format for a New Featurette

Here is an example of a new featurette:

```
CSCdm08789:New Record
-----
Headline:Add control policy for giaddr used in relayed cable DHCP requests
Status:A
To-be-fixed:12.0T
DE-manager:groeck
Enclosure-count:1
Modified-file-count:0
```

36.5.2 Email Format for a Modified Featurette

Here is an example of a featurette DDTs changing state from “A” to “R.”

```
CSCdm04971:
=====
Old Record
-----
Headline:Need set trunk e1/t1 hidden command for Octal PRI card
Status:A
To-be-fixed:12.0 12.0T
DE-manager:rmeadows
Enclosure-count:1
Modified-file-count:0

Modified Record
-----
Headline:Need set trunk e1/t1 hidden command for Octal PRI card
Status:R
To-be-fixed:12.0 12.0T
DE-manager:rmeadows
Enclosure-count:2
Modified-file-count:9
```

36.6 Process Overview

The following is a description of the featurette process.

- At concept commit, if the functionality appears to be a featurette, a DDTs is opened in CSC.sys as featurette (N state) and the commit-type is set to featurette.
- DE-manager assigns development staff and identifies project team as needed (A state).

- DevTest and Development managers assign test personnel to execute test plan (including EFT if required by DE manager).
- Responsible engineer writes a Functional Specification and develops the featurette (O state).
- Once the target release is identified, DE-manager completes the **partial commit requirements form** and acquires the official marketing name from their marketing organization.
- Once the functionality is developed, the project is re-assessed to determine if the new functional still qualifies as a featurette or as a feature. If the scope of the project has gone beyond the bounds of a featurette, then the functionality must go through the feature commit process.
- Responsible engineer gathers code review approval and adds the code reviewers to the DDTs. Code review issues should be documented as an enclosure to the DDTs.
- DevTest engineer authors Test Plan (manual or automated, addressing per release issues as necessary). The test plan must include REGRESSION TESTING as well as FEATURE TESTING.
- DE and DevTest managers approves Test Plan.
- The DE manager updates the Commit Requirements Form with the details about converting the feature testing into an automated script.
- Once the development code has been parented to the PI branch that it plans to commit into and has completed a recent sync, images will be build for testing.
- DevTest staff execute test plan.
- All defects found during regression and feature testing are then resolved prior to the commit of the featurette.
- DE-manager ensures that the code is complete, all code review issues have been closed, all testing/re-testing has been completed on the correct branch on a recent sync point, all defects associated with the featurette have been resolved, and resolves any other open issues from the team or groups such as parser-police and mib-police.
- DE-manager identifies and notifies feature module writer or ITD Technology Owner using the URLs on the Commit Requirements Form to identify the correct person.
- Dev-engineer commits featurette into targeted PI branch and resolves DDTs (R state or M if initial commit is incomplete).

36.7 Documentation

Customer documentation is required to provide customers with information about new Cisco IOS functionality and how to implement it. If customer documentation is not available, customers have no other way of accessing this information. Customer Documentation will only document featurettes with a Commit Requirements Form as follows:

- Featurettes will be documented in IOS Configuration Guides, IOS Command References, IOS Feature Modules, and Cisco IOS Release Notes.

36.7.1 Featurette Information From DDTs

Information on featurettes in DDTs can be found at the following URL:
<http://wwwin-release.cisco.com/rellops/Featurette/index.html>

Current Cisco IOS Initiatives

37.1 Overview

Cisco IOS DEs are expected to develop code according to a certain style (for example, see Appendix A, “Writing Cisco IOS Code: Style Issues”) as well as standards, procedures, and initiatives related to programming at Cisco, such as:

- SW Development Phase Containment Guidelines—Described in this chapter.
- Carrier Class Initiative—TBD
- Future Quality Initiatives—TBD

Note Cisco IOS development quality questions can be directed to the interest-os@cisco.com alias.

This chapter includes information on the following topics:

- Phase Containment Guidelines
- Carrier Class Initiative Guidelines - TBD

37.2 Phase Containment Guidelines

Software development phase containment guidelines were created to drive consistent, scalable processes that enforce effective static analysis, code reviews, and unit tests across CDO. The goals of these processes are to:

- Deliver fewer bugs from Development to DevTest.
- Focus our tools enhancements into a common and more capable tool set.
- Take corrective action in Development before handing code to DevTest.

In order to succeed with the Phase Containment Guidelines, the development community at Cisco was asked to:

- Follow the processes that were designed by the Cisco Engineering DEs, for the DEs, and owned by the DEs to assure that they are relevant and helpful.

Note This was implemented by having the DEs work with Engineering Education to create the *Phase Containment for Development Engineers* course and having the BU technical leaders responsible for teaching the *Phase Containment for Development Engineers* course to the DEs in their respective BU (ask your BU management to sign up for this course).

- Use the supported tools for efficient code development and testing.
- Assume local ownership of procedures and results by having BU technical leaders train managers and DEs.
- Measure both adoption and effectiveness at DE-Mgr level to demonstrate progress (however, measuring effectiveness is not mandatory at other levels).
- Complete the Phase Containment pilot in IOS/X, VTG, NMTG, BEMR, and ONG to assure that it scales to all of CDO.

Thus, all BU DEs are asked to complete the following tasks:

- Static Analysis (SA)—Run static analysis tools for all features, featurettes, and fixes (not focusing on legacy warnings at present). It is also recommended to run static analysis tools for ports, merges, and syncs.
- Code Review (CR)—Complete code review for all features, featurettes, and fixes.
- Unit Test (UT)—Unit test all features, featurettes, and fixes (IOS only).

The requirements for the completion of these tasks are described in the following subsections. Along with SA, CR, and UT, all BUs are asked to complete Phase Containment metrics. The main mechanism to determine the success of Phase Containment is through Development Phase Escape Analysis, and the key metric is EDP (Escape Detection Process). EDP shows if Phase Containment is making a difference by taking the number of bugs found downstream that should have been detected in Development, and comparing the number to the baselines created during Phase Containment pilots. For more information on EDP, see: <http://swops.cisco.com/page/edp.html>. For detailed information on phase containment specifically for your BU, take the *Phase Containment for Development Engineers* course offered by your BU. To learn more about Phase Containment, see: <http://wwwin-engd.cisco.com/info/pc/phaseContain.html>. See also the CDO “*Development Phase Containment / Software Development Practices*” web page at: http://swops.cisco.com/page/swops_engprac_pc.html.

37.2.1 Static Analysis Requirements

The requirements to complete the static analysis are as follows:

- 1 Reduce the cost of finding common programming errors by running the suite of static analysis tools *before* committing the code.
- 2 Use `cstatic` or `prep_commit` to run the static analysis tool suite.
- 3 If required, mark any warnings that are either noise from the tool or accepted coding within the BU group.
- 4 It is highly recommended that new warnings identified by static analysis be removed prior to CR.
- 5 It is highly recommended that warnings marked as noise be reviewed during CR.
- 6 Remove all newly identified valid warnings prior to code commit.
- 7 Before code commit, static analysis tools should generate zero warnings.

- 8 When resolving a DDTs that should have been found by SA, mark the “Reason” field (this field is available at the time of this writing, but is planned to be renamed the “dev-escape-phase” field) as “static-analysis”.
- 9 Ensure that the `Static-analysis-runinfo-branch` (used for Code Reviews) and the `Static-analysis-branchinfo` (used for Adoption Metrics) are generated. These are automatically generated by the `prep-commit` tool (when run with `-bugid`), via a link in the static analysis log provided by the various `cstatic` commands when using `html`, or via the tool documented at <http://wwwin-ses.cisco.com/staticAnalysis/attachment.html>

Note The cctools are being modified to use a cookie generated by the static analysis toolset to automate the collection of adoption metrics – when this change to cctools is in place, there will be no need to semi-manually or manually generate attachments.

37.2.1.1 Static Analysis Resources

Static Analysis Tools:

<http://wwwin-ses.cisco.com/staticAnalysis/>

Static Analysis EC, EDCS-365726 at:

http://wwwin-eng.cisco.com/Eng/IOS/IOS_Quality/Sa_Funct_Team/Quality_Records/EC_for_SA.ppt@latest

37.2.2 Code Review Requirements

The requirements to complete the code review are as follows:

- 1 Use the Code Review Process Handbook (EDCS-347748) and the Code Review Checklist (EDCS-359372).
- 2 It is recommended to perform a basic code review via email for low risk code; other code should usually have a formal review. The degree of formality is based on factors such as the complexity of the code, the risk level of the code, and the location of reviewers.
- 3 In a formal code review, a programmer other than the author must understand the code and present it to the team.
- 4 Each member of the formal code review team can look for one or more specific assigned categories of faults from the CR (Code Review) checklist, but at least one reviewer must review the code from an overall perspective.
- 5 The formal code review team must review the unit test plan and any ignored static analysis warnings.
- 6 If desired, the formal code review team can use CRRQ+ (Code Review Request Queue) or PRT (Peer Review Tool) to coordinate and record the code review. Otherwise, the formal code review team must create DDTs enclosures that meet the following conventions:

- (a) Naming Convention (Where `branchname` is the name of the branch where the code is.):

`branchname-code-review`

- (b) Formatting Conventions:

For Adoption Metrics, include within the enclosure:

`++Status: Closed`

For Effectiveness Metrics, also include within the enclosure (Where # is the number of defects.):

```
++High Defects: #
++Medium Defects: #
++Low Defects: #
++Defects: #
```

- 7 When resolving a DDTs that should have been found by CR, mark the “Reason” field (this field is available at the time of this writing, but is planned to be renamed the “dev-escape-phase” field) as “code-review”.
- 8 Begin posting CR data for Adoption and Effectiveness Metrics:
 - For CRRQ+ this is automatic for Adoption Metrics, but for Effectiveness Metrics, the queue’s Local Admin must add “yes” to MetricsCapture in the CRRQ config file (some queues may also want to add “yes” to DDSTUpdateReview in the CRRQ config file so the DDTs code reviewer field will be updated).
 - For PRT this is automatic if the review includes the DDTs# & review type = Code.
 - For DDTs, see 6 (b) above. Although not required, it is highly recommended to also post the number of CR issues per KLOC (1000 lines of code) reviewed (this is done automatically when using CRRQ+ and PRT).

37.2.2.1 Code Review Resources

Code Review Checklist (EDCS-359372) at:

http://wwwin-eng.cisco.com/Eng/Quality/Cisco_Qual_Initiative/Software_Track/Development/ReviewChecklists.doc

Code Review Request Queue at:

<http://wwwin-crrq.cisco.com/crrq-admin.cgi>

Peer Review Tool at:

<http://wwwin-tools.cisco.com/cse/PRT/home.do>

Code Review EC, EDCS-372952 at:

http://wwwin-eng.cisco.com/Eng/Quality/Cisco_Qual_Initiative/Software_Track/Development/EC_for_CR.ppt@latest

37.2.3 Unit Test Requirements

The requirements to complete the unit test are as follows:

- 1 Develop the unit test strategy and:
 - (a) For bug fixes, add a Unit-Test attachment (or Unit-Test-branch_name) to every resolved DDTs/CDETS. See the *Unit Test and Unit Test Integration Guidelines* (EDCS-373001, Section 6.1).
 - (b) Use the *Software Unit Test Plan* (EDCS-373002) template for developing feature test plans (this can also be done optionally for featurettes).
- 2 Design and develop the unit test cases.

- 3 Perform the unit test. Improve the breadth and depth of unit testing (see the *Unit Test and Unit Test Integration Guidelines* for more details):
 - (a) To reduce the unit test defect escapes:
 - Use white box test techniques and tools [such as the SESUT (Software Engineering Solutions Unit Test) Test Harness] to find more and different kinds of bugs, and to validate how the code works [SESUT is integrated with CFLOW and MIF (Middleware Injection of Faults)].
 - Increase the breadth of black box testing to find bugs earlier and to increase overall productivity. For example, you can use Expect or TCL for black box testing to verify that the fix/feature works as intended.
 - Increase SRT (Selective Regression Testing) to verify that a feature/fix does no harm.
 - Review the UT escapes to continuously improve testing.
 - (b) Atomic Unit Test: For the smallest testable pieces of code.
 - (c) Unit Integration Test: For new test cases, used to cover the interaction between LDUs (Logical Design Units).
- 4 Assess the unit test results and determine the actions to be taken.
 - (a) When appropriate in the development cycle, file a DDTs for each bug found in UT with “FOUND = unit-test”.
 - (b) When resolving a DDTs that should have been found by UT, mark the “Reason” field (this field is available at the time of this writing, but is planned to be renamed the “dev-escape-phase” field) as “unit-test”.

37.2.3.1 Unit Test Resources

Unit Test and Unit Test Integration Guidelines (EDCS-373001) at:

http://wwwin-eng.cisco.com/Eng/Quality/Cisco_Qual_Initiative/Software_Track/Development/UTguidelines.doc

Software Unit Test Plan (EDCS-373002) at:

http://wwwin-eng.cisco.com/Eng/Quality/Cisco_Qual_Initiative/Software_Track/Development/Unit_Test_Plan.doc

SESUT at:

http://wwwin-ses.cisco.com/unittest/sesut_ios.html

Selective Regression Testing (SRT) of IOS Software - User Guide at:

http://wwwin-ses.cisco.com/testing/srt_userguide.html

Unit Test EC, EDCS-372576 at:

http://wwwin-eng.cisco.com/Eng/Quality/Cisco_Qual_Initiative/Software_Track/Development/IOS-X_Unit_Test_EC.ppt@latest

SES Homepage (links to CFLOW, MIF, others) at:

<http://wwwin-ses.cisco.com>

Appendices

Writing Cisco IOS Code: Style Issues

Added information about a code indentation checking tool in a new subpara “Checking Indentation,” in sub-section A.4.1, “Specific Code Formatting Issues.” (January 2011)

Added the command length details as a new subpara “Command Length”, in sub-section A.4.1 “Specific Code Formatting Issues”. (July 2010)

Added a note on __func__, __FUNCTION__ and __PRETTY_FUNCTION__ to the “Function Prototypes” sub-section of A.3.2 “Fifty Ways to Shoot Yourself in the Foot”. (July 2010)

Added details on using return statement with parantheses to A.4.1 “Specific Code Formatting Issues”. (June 2010)

Added a note on exception to explicit typecasting to A.3.2 “Fifty Ways to Shoot Yourself in the Foot”. (March 2010)

A.1 Purpose of This Appendix

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.

Note Cisco IOS code style questions can be directed to the ios-style-guide@cisco.com, software-d@cisco.com, software-questions@cisco.com, and miss-manners@cisco.com aliases. [software-questions](mailto:software-questions@cisco.com) should be used to ask non-general questions. Questions of general interest should be sent to [software-d](mailto:software-d@cisco.com). ALWAYS do what research you can before sending email to either [software-questions](mailto:software-questions@cisco.com) or [software-d](mailto:software-d@cisco.com). It’s a good idea to mention the research you’ve completed when asking the question.

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
- Coding for Scalability

Note For questions on coding for serviceability, contact ios-serviceability@cisco.com.

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 Chapter 20, "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 Abuse the Pre-Processor

When defining constants, they should be typed explicitly to match the type of variable that they will be assigned to, and/or compared to, or otherwise manipulated with. For example, a timeout value that is always assigned to or compared with a ulong variable should be specified as `UL`. Similarly, bitmasks variables and constants should always be specified as one of `uint`, `ulong`, or `ulonglong` (`U`, `UL`, `ULL`) as required.

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
```

Enumerated Types 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
{
    A = 1, /* skip zero */
    B,
    E = 5,
    F,
    G,
    X = 24,
    Y,
    Z
};
```

If you must use the #define idiom, write the #defines 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)
```

Avoiding Unlabeled Constants

When a particular value is reused in multiple places, and there's any possibility that the value might change, create a symbol for it (be it a #define, const, enum value, or whatever). Even if a value isn't likely to change, or isn't used in multiple places, you should usually give it a name (using #define or const) so that the meaning of or the reason for the value is clear. Exceptions to this rule include occasions when the inherent properties of the value itself are involved, such as using 10 or 2 as a radix in numeric conversions.

Here is a code example that should have a labeled constant:

```
NUMBER(...,
        OBJ(int, 2), 0, 60,
        "Keepalive period in seconds (default 10)");
```

It should be something like this:

```
#define stringof(x) NAME_TO_STRING(x)

/* in some subsystem header file */

#define XYZZY_MIN_KEEPALIVE 0
#define XYZZY_MAX_KEEPALIVE 60
#define XYZZY_DEFAULT_KEEPALIVE 10
```

Then you can write:

```
NUMBER(...,
        OBJ(int, 2), XYZZY_MIN_KEEPALIVE, XYZZY_MAX_KEEPALIVE,
        "Keepalive period in seconds (default "
        stringof(XYZZY_DEFAULT_KEEPALIVE) " )");
```

This example will generate:

```
"Keepalive period in seconds (default 10)"
```

Remember, don't apply these rules slavishly, just for the sake of satisfying some guideline. Constants should be given names when there is some manifest value in doing so, for example, simplifying maintenance of the code, or improving readability. But there are also plenty of cases where "a zero is just a zero" and you should use some judgement between whether a special name is warranted, or you should just use the constant directly.

Consider:

```
NUMBER(...,
        OBJ(int, 2), 0, 60, ...
```

versus

```
NUMBER(...,
        OBJ(int, 2), 0, MAX_KEEPALIVE, ...
```

versus

```
NUMBER(...,
        OBJ(int, 2), MIN_KEEPALIVE, MAX_KEEPALIVE, ...
```

The upper limit may well be changed over time. Defining a constant such as MAX_KEEPALIVE for the value 60 makes good sense. However, the lower limit of zero is unlikely to ever change. It is just an intrinsic lower bound because entering a negative number wouldn't make sense for this value, not [presumably] because we expect to have any other minimum bound. Putting a symbolic name for that zero doesn't really buy you much, and in fact makes things harder on the reader because now you have suggested to the reader that the value might be mutable, something other than the obvious zero. Additionally you've potentially wasted the reader's time by requiring the reader to look up a value just to see that it is the zero she/he would have expected all along. Don't make the reader work for something when there isn't a purpose for it.

So, in this particular example, the second option is probably the best way to do things.

A.2.1.1 Avoid Conditional Compilation

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
#define 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.

For example, using “#if defined(AS5400) || . . .” is the worst of all possible solutions, and there are better alternatives. Also, the Gnu **autoconf** tool generates makefiles with **-DHAS_XXX** to enable certain generic features, and then builds a profile per platform (or environment) based on which capabilities need to be turned on.

Conditional compilation is problematic at a number of levels, and first and foremost among them is infection. That is, when using a preprocessor conditional inside a public header file, all includers of that header file become “infected” with the conditional compilation. Conditionals in public headers generally have negative consequences. There are two large problems:

- 1 Includers of a public header become “infected” with the conditional compilation.

In effect, they must be compiled/maintained/managed for each state that the conditional can be defined to be.

- 2 Similarly, conditionals introduce more makefile churn and likely more object directories, which means longer build times for everyone.

Although conditionals inside private headers and .c files can be appropriate, they should be used as a technique of last resort. In addition, there is a special type of .c file for which conditionals should be avoided: “platform-generic” files. A platform-generic file is a .c file that looks like it is platform-independent but turns out to have per-platform dependencies. This means that the same .c file will be compiled for each platform. In many cases, using conditional compilation is unavoidable, however there are ways of coding to make it more clear to future maintainers of the files. For example, you can “pull” the template. The idea is that per-platform (or even per-feature) definitions are placed in a per-platform header file and the “template” code is placed in a generically named file. Each platform includes the per-platform definition and the template into an appropriately named platform source file.

We cannot move away from all platform-specific conditionally compiled code because if we do, we will have code duplication as a result. So, the platform-generic template helps us achieve a source code solution in an explicit manner. We want to do this as little as possible, and link-time and run-time solutions are preferred (even encouraged), but they are not always the correct choices.

Sometimes, compile-time solutions are the best and creating an explicit mechanism will help us manage unintentionally broken builds involving platforms. Moreover, it reduces the number of maintenance points associated with conditional branching of the code. It's better to have one file contain the a set of templates rather than many files contain “ifdefs”. In these cases, be sure to include comments that explain why conditional linkage and/or run-time checking is not applicable and why conditional compilation is the *only* viable solution.

For example, here is a per-platform header file named `c7100_def.h`:

```
#define MYDEF <something>
```

Here is another sample per-platform header file named `rsp_def.h`:

```
#define MYDEF <something-else>
```

Here is a sample `generic_platform.template` file that is used by both platforms above:

```
/*
 * this file cannot be compiled on its own,
 * it must be included inside a .c file
 */
void generic_func (void)
{
    ... MYDEF ...
}
```

Include the following in `c7100_something.c`:

```
#include "c7100_def.h"
#include "generic_platform.template"
```

Include the following in `rsp_something.c`:

```
#include "rsp_def.h"
#include "generic_platform.template"
```

Note The template file is in effect part of the “public” API. This way we maintain generic template files that will work across all platforms assuming the appropriate conditionals are defined. This also help future developers “know” the consequences of modifying the template file, that is, that it will affect all platforms.

There are times when the variable or function that you're trying to bracket simply won't exist for a given platform or image set. Yet conditional compilation can be useful; however, it should be considered a choice of last resort, and the principle should be to follow along functional lines, rather than platform lines.

A much better solution is to use conditional linking. Of course, in this case a run-time check would be appropriate. For example, use a variable that is assigned at init time, as shown below:

```
platform.h
-----
extern boolean platform_ecc_support;

as5400_something.c
-----
boolean platform_ecc_support = TRUE;
```

There are cases of conditional compilation where we have the following:

```
#if defined(AS5400) || defined(MARVEL) || defined(C7200) || ...
    printf("ECC enabled on bank %d\n", i);
#endif
```

This churns the code enormously in a lot of different places, since every new platform has to propagate its presence into various source files in the current usage model. So, rather than doing conditional code based on individual platforms, you can do it based on platform capabilities and attributes, as shown here:

```
#if defined(PLATFORM_HAS_ECC)
    printf("ECC enabled on bank %d\n", i);
#endif
```

This reduces the amount of churn because then we don't change source files, and we only need to change the `machine/` files, such as:

```
==== machine/cisco_as5400.h ====
...
#define PLATFORM_HAS_ECC 1
...
```

A.2.1.2 Guidelines for Using the ## Operator

The use of the `##` operator should be avoided unless there's a good argument for doing it, as in the case with parser macros. With parser macros, the “`##`” makes it possible to make macros that build data structures for the parser to follow. Without “`##`,” it would be very hard to write macros to do this. The resulting naming conventions are far too complex for most of us to follow rigorously.

Here is an example of the use of the `##` operator that seems like a good idea but isn't:

```
#define DECLARE_HANDLER_REGISTRAR(foo) \
    void register_##foo ( context_t *ctx, handler_##foo##_t handler) \
    { \
        ctx->reg.foo = handler; \
    }
...
DECLARE_HANDLER_REGISTRAR(event1)
DECLARE_HANDLER_REGISTRAR(event2)
DECLARE_HANDLER_REGISTRAR(event3)
...
```

The problem with this example is that tools like `cscope`, `gid`, and `tags` can't find the symbol's definition. Using the `##` operator seems like a good idea because it enforces a naming convention. Enforced conventions are a good idea, and naming conventions are a great idea. But the disadvantages outweigh these advantages. It would be better to *comment* the naming convention and just follow it—after all, how often are new instances added? Probably pretty rarely.

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 grafting 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.2.6 Stack Size

It's not a good idea to declare large automatic variables for all Cisco IOS coding because they can overflow the Cisco IOS process stack (recursive algorithms can also overflow the process stack). In Cisco IOS, a process's stack is fixed in size and cannot grow. A stack frame larger than 1024 bytes will cause a static analysis warning.

A.2.7 Function or Macro or Inline Function?

Sometimes it is hard to decide whether to use a function, a macro, or an inline function. Here are some guidelines.

A.2.7.1 Reasons to Use Macros

Macros may be preferred when portability is one of your primary concerns. If you are writing code where you do not want to use a real function, but you want the code to look like you are using a real function, then macros are your only choice if your compiler doesn't support `inline`.

We all know that function-like macros are not perfect substitutes for real functions; that's why we also have that other little convention of writing function-like macro names in ALLCAPS to alert the coder to be careful when using that "function".

Macros are great for defining constants or short-handing complex expressions or oft-used simple expressions. In recent times, macros are also great for gluing together string tokens.

A.2.7.2 Should This Be a Macro?

The following example answers “no” to every question below, so it’s pretty clearly a negative example. It is an example where you have to validate a buffer write (for filling a packet), to ensure that it does not spill beyond the boundary. You need to do this even if you are writing a single byte to the buffer.

```
#define XXX_WRITE_BYTE(a) \
    (remaining_length >= 0 ? ((*dataptr++ = (a)), --remaining_length,
    TRUE) ?      FALSE)
```

You cannot use an inline here since what you are trying to do is:

```
if (!XXX_WRITE_BYTE(a))
    return;
```

You may ask the following questions:

Is it to provide a general interface for controlled writing to a buffer? No, because the names of the buffer and counter variables are hardcoded into the macro without any concept of scope, and it assumes all callers who would try to overfill a buffer want to return without a return value on failure—not a general solution.

Is it readability? No, because the proposal acknowledges that the “return” behavior is an exception to standard coding guidelines and practices, and therefore something a developer would not expect from such a construct.

Is it microoptimization without profiling? One engineer did a quick test of a macro not unlike the one proposed and a static function (without even using the `inline` keyword) that returned `TRUE/FALSE` and where the caller would do `if (!X(a)) return;`. The Sun Solaris compiler with full optimization generated identical code for the function and the macro, but the function version made clear in the calling code what was being attempted, that it could fail, and what the consequences of failure were. In fairness, the old GCC we use didn’t optimize the function version as well, but compilers do improve over time, and unless you can show a *system* level performance improvement through actual profiling, such microoptimizations make for higher support costs and offer nothing in return.

Is it to save typing? To paraphrase the different versions of the quote attributed to Albert Einstein, “make everything as simple as possible but no simpler.” Reducing lines of code and reducing repetition can simplify the reading of the code, but if something important like a “return” becomes obscured, you’ve gone too far and there will be a real cost to the people who have to maintain this code after you have moved on to other things.

There are too many macros that fail the tests above, making the code more cumbersome, and at a micro level, measurably slower. Macros aren’t always bad, but you should always ask “what is it about the base language that prevents me from solving the problem without a macro?” and “is the gain greater than the cost?” while trying to be realistic about what the real gains and costs are. Saving a millisecond in a one minute boot sequence is zero gain even if you are saving a million cycles, and a “clever” construct that may seem obvious to an author who has had months to design a feature may not be obvious to any of the random developers on any of N platforms who may have to debug the code without any background on that feature.

A.2.7.3 Pitfalls of Macros

Rather than clinging to a rule about avoiding macros, it’s better to understand the problems with macros and try to avoid them—either by avoiding macros when there’s a better way, or by minimizing the pitfalls.

We generally expect macros to behave like functions. Any macro that doesn't add to confusion when reading the calling code. The fact that a macro returns causes a readability problem. This problem could be alleviated by changing the name to "XXX_WRITE_BYTE_OR_RETURN", or something like that, to give the reader a clue. Otherwise, someone might add a `malloc()` at the beginning of the function and `free()` at the end without noticing that it has many early returns.

Another case is when you have "hidden parameters" that are best to avoid in order to make the code as easy to read as possible. You should include the length and pointer as macro arguments.

Here are some pitfalls in using macros:

- 1 Strive to make your code easier to read, rather than easier to write. A common pitfall is that we often use macros to make it easier to write. Instead we should use them to make it easier to read.
- 2 Macro calls in executable code look like function calls. Whenever engineers write them not to act like functions, it makes code harder to read. This can often be overcome by giving clues, like carefully choosing names and parameters.
- 3 Macros containing `return`, `break`, `continue` or similar statements affecting the outside scope should make that clear from their names and use in the calling code. Macros can be defined that really mess with C grammar; it's best to avoid this completely for the obvious reason.
- 4 Macros often violate the normal association of clauses to `if/else`.

Example:

```
#define X(a) if (a) then xxx  
...  
if (foo)  
X(q);  
else  
YYY;
```

The programmer presumably wants `yyy` to be executed if `foo` is false, but it will be executed if `a` is false. The good news is that this problem is eliminated if everyone calling the macro follows the Cisco convention to always use braces. But it's best not to write macros that rely on other people following conventions. Instead, use

```
#define X(a) if (!(a)) then ; else xxx
```

Here is another example:

```
#define X(a) do { if (a) ...; } while(0)
```

`do { blah } while(0)` is a standard technique for safely embedding code in such a way that it looks like a simple function call.

Then someone adds a comma operator to do "just one more thing":

```
if (foo)  
X(a), b();  
else  
c();
```

and everything falls apart, as the macro `X(a)` is a statement, not an expression like a function call.

- 5 Long macros are hard to debug using a debugger because you can't step through the definition lines.

- 6** Macro formal parameters should be enclosed in parens in the macro definition to avoid unintended association of operators. Likewise, a macro that produces a value should always be enclosed in parens. For example:

```
#define FOO(a, b) a * b
```

is wrong if you call it as FOO(i+1, j). Instead, use

```
#define FOO(a, b) ((a) * (b))
```

Remember, the entire macro expansion should begin and end with parens if it's a complex arithmetic expression. On the other hand, *don't* put parens around simple values, like 15000, since you might want to stringify that quantity when building a help string. Here is an example:

```
#define MAXTIMEOUT 15000
```

```
NUMBER(x_name, x_accept, x_alt, OBJ(5,int), 1, MAXTIMEOUT,
       "Link quality timeout [range 1..." stringify(MAXTIMEOUT) "]", ...);
```

- 7** With macros, you don't get type checking like you do with functions.
- 8** Macros are type insensitive. This is usually a bad thing, because the writer of the macro usually had specific types in mind when he/she wrote it. GCC 3.x has some extensions that load type-flexibility into macros but this gets hard, fast (it's useful if you've a code-generator machine which spits out same macro-name for different types of situations).

Use a `static inline` function if possible. If it's a long function, just use a normal function -- that's what they're for!

On the other hand, if a constant is to be used largely with variables of a particular type (assignments, comparisons, arithmetic, etc.), the following avoids static analysis warnings about comparison of signed and unsigned quantities:

```
typedef ulong timeout_t;

#define INITIAL_TIMEOUT    100UL
#define MAX_TIMEOUT        5000UL

timeout_t n1 = INITIAL_TIMEOUT;

...
if (n1 <= MAX_TIMEOUT) {
    ...
}
```

On the other hand, you've lost the ability to stringify the constant and use it in messages.

- 9** You have to be wary of having parameters appear more than once in the expansion. For example:

```
#define max(x,y)   ((x) >= (y) ? (x) : (y))

uint a, b, c;

a = 3;
b = 2;
c = max(++a, b);
```

You'd expect `c` to be 4, but it's actually going to be 5.

- 10** Don't terminate macros with a semi-colon.

A.2.7.4 More About Inlines

The general points are:

- 1 Don't assume inlines are more efficient than functions.
- 2 Inlines are generally better than macros because of type checking.
- 3 It shouldn't be necessary to force the compiler to inline a function (when the definition is in a .h file or the same .c file.)
- 4 The difference between `static inline` and `extern inline` is that `extern inline` means that the definition that is provided is used *only* for inlining. No real function is ever created from this code.

From ISO/IEC 9899:1999(E), section 6.7.4:

A file scope declaration with “extern” creates an external definition.

The context of section 6.7.4 is the `inline` keyword, and ANSI/ISO C defines behavior that is not compatible with current GCC hacks, which even the GCC folks are moving away from.

`extern inline` is thus the closest analog to a preprocessor macro. If you use the function name in a way that can't be inlined (such as attempting to take the address of a function), you get a compile error, just as you would if using a preprocessor macro. `extern inline` forces the function to be inlined, and a callable version of the function to be generated. Since inlines rarely have both a separate declaration and a definition (just the latter), you'll most likely never be in a position to call out to an inline like a regular function.

You would need a header file like:

```
#if defined(LOW_END_PLATFORM_OR_PERFORMANCE_NOT_CRITICAL)

extern void my_inline_func_prototype (const char *name, ipfib *ipf);
...

#else

extern inline void my_inline_func_prototype (const char *name, ipfib *ipf)
{
    ...
}

...
#endif
```

`static inline` means that if the source file uses the function name in a manner that can only be satisfied by a reference to a real function (for example, getting the function address), then the compiler automatically generates a real function within the object file and uses that real function in places where it cannot substitute the inline definition. This function is generated as a `static` function, so its scope is limited to the object file in which it was placed.

For our purposes, `extern inline` is almost always what you want to be using. One sees `static inline` a lot in source files but that seems to be because of ignorance.

From the GCC documentation for many release cycles:

(If you are writing a header file to be included in ISO C programs, write `_inline_` instead of `inline`. See Alternate Keywords.) You can also make all “simple enough” functions inline with the option **-finline-functions**.

...

Since GCC eventually will implement ISO C99 semantics for inline functions, it is best to use `static inline` only to guarantee compatibility. (The existing semantics will remain available when `-std=gnu89` is specified, but eventually the default will be `-std=gnu99` and that will implement the C99 semantics, though it does not do so yet.)

and later under “Alternate Keywords”:

`-ansi` and the various `-std` options disable certain keywords. This causes trouble when you want to use GNU C extensions, or a general-purpose header file that should be usable by all programs, including ISO C programs. The keywords `asm`, `typeof` and `inline` are not available in programs compiled with `-ansi` or `-std` (although `inline` can be used in a program compiled with `-std=c99`). The ISO C99 keyword `restrict` is only available when `-std=gnu99` (which will eventually be the default) or `-std=c99` (or the equivalent `-std=iso9899:1999`) is used.

The way to solve these problems is to put `_` at the beginning and end of each problematical keyword. For example, use `_asm_` instead of `asm`, and `_inline_` instead of `inline`.

Clearly it is the intent of the GCC folks to go with the ISO standard semantics, and ultimately when you want to use certain C99 semantics, you have to bring in the ISO `inline` semantics at the same time. Let’s not make that impossible.

`static inline` does the job most of the time, and for the cases where someone is taking the address of the function, one should examine if the function should be an inline at all. Indeed, everything we’ve encountered as inlines where we’ve had to notice that detail has provided no benefit at all, and they have racked up real costs in extra maintenance (think adding lots of unrelated inlines to a header everyone includes, then not being able to include every other header those inlines need, and then externing in global variables for the inlines to use and not type check against their actual definition.) Bad coding practices are like potato chips—most people can’t have just one.

For the few who can’t resist the compulsion to write non-standard code, use the GCC `_inline_` keyword for GCC-only semantics.

Normally we do not want the compiler to autogenerate real functions, but instead explicitly create real functions ourselves (using a different name for the real versus the inline) where they are needed. That is to say, we do not want to be using `static inline`.

Our usual naming convention is that any inline functions we create has names ending in `_inline`. Thus when we have a routine for which we’ve created both an inline and real variant, they have names like `foo_inline` and `foo`, and there is no ambiguity about whether `foo` refers to an inline or a real function at any spot in the code.

Having two different distinct function calls (one representing an inline and another for those who need a real function) is sometimes a bit awkward, but has the advantages that

- (a) It is obvious where an inline is being incorporated into another routine - no one will accidentally bloat a routine by inlining something unintentionally. Developers must consciously choose to use one variant or the other.
- (b) Code won’t bloat because the compiler produces multiple redundant `static` versions of the function scattered across multiple `.o` files. That is, if files `a.c`, `b.c`, and `c.c` all included a header file that had a `static inline foo()`, there would be a possibility for three real, statically scoped, copies of `foo()` to be created. Once a real function is needed, it makes more sense to just create one copy of it, globally scoped, that all the `.o` files can use.

Of course, there are exceptions to every rule, and the convention of appending `_inline` to our inlines isn't well adhered to. In particular, many of our functions that are really inlines that provide a sugarcoated syntax to fetch a field out of a structure often lack the `_inline`. Whether or not this is a good practice, and whether or not true macros should be used instead for this type of thing are the subject of religious arguments.

- 5 Mind the compile time cost of including complex code in popular include files.
- 6 Consider putting inlines in their own header files.
- 7 In-lining bloats the code. One exception is that an inline that handles many different situations (for generality), but the situation (codepath) is known in advance, can generate very small code and preserve readability.

For instance,

```
static inline void handle_my_error (enum my_errs errno)
{
    switch (errno) {
        ...
        case ERR_NONE:
            break;
    }
}
```

and then:

```
handle_my_error (ERR_SOME_VAL);
```

should have all of the unreachable code elided, resulting in very little code indeed. And the case of:

```
handle_my_error(ERR_NONE);
```

should result in no code at all.

Having a set of inlines that are always expanded in place (as they are) without regard to whether the function call overhead is acceptable or not, may have an unexpected impact on global system performance—the extra memory required by the inlined functions could be enough to push useful functions (which aren't inlined) out of the instruction cache, slowing performance.

The above example only works for cases where at least one parameter is a constant that determines a major aspect of the code path through the inline function. A common example of this is populating a state machine where one of the parameters is the next state, or an error condition.

- 8 Inlines are type-checked. This is usually a good thing, but occasionally a bad thing.
- 9 Inlines are flexible with respect to expansion—depending on the compiler options, they can be made to go inline or out-of-line. Macros are always inline and so cannot be fashioned to go low-end/high-end/whatever.

As a side-note, this is true, but largely theoretical today. We have not attempted to control inline expansion on the basis of platform capabilities to any extent.

- 10 Inlines are type-checked. This is usually a good thing, but occasionally a bad thing.
- 11 Inlines should be avoided unless a strong argument can be made that they improve performance (and this argument has to take multi-platform issues into account as appropriate). Small wrapper and accessory functions can be defined in header files as static (without `inline`), and the compiler is free to optimize the code.

A.2.7.5 Some Last Comments about Inlines and Macros

There is another layer about physical partitioning that is worth addressing. The fact that the inlines and macros that are very rarely used are ending up in commonly included header files is making for more build time overhead and namespace pollution. People need to think through where they are placing this code, and if necessary, create new header files. We could also use better naming conventions because there are a mass of macros with one and two word names that could mean any number of things in any number of contexts.

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 editing buffer? 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 Standard
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-enged.cisco.com/common/doc/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. Whenever you create a new function, you must create a function prototype in a location that is appropriate for the scope of the function. If the function is part of a component's API and is expected to be used by code outside the component (global scope), then its prototype must be placed in a public header file (`foo_public.h`, or in some cases, simply `foo.h`). If the function will be used by code in multiple source files within a single component (component scope), then place its prototype in a component-private header file (`foo_private.h`). If the function is used only within the file where it is defined (file-local scope), then it should be declared static, and the function definition can also serve as the prototype if the code is arranged so that the definition precedes all uses. If for some reason this function ordering is not possible, then place the prototype for the static function at the head of the file (note that this is the *only* situation where it is allowed to place a prototype in a `.c` file - for a static function defined and used only in that file).

If the usage of a function changes scope (usually moving from static to component-private, or from component-private to public), then the prototype definition must be moved accordingly.

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. All code that uses a function (including the file that defines it!) must include the one header file that contains the function's prototype. Do not copy or clone the prototype into some other header file for "convenience". If you find yourself coding a prototype for some other component's function, that is a major clue that something is wrong. If the component has provided a public API function that you need, but no header file with a prototype for that function, write a DDTs against that component. Do *not* just stick the prototype you need into some header file of your own; it is not sufficient that a prototype merely exist somewhere, in some header file - it must be defined exactly once, in the correct header file.

Attributes can be attached to prototypes. If the function is like `printf()`, use the `format` attribute (the `format` attribute specifies that the function takes `printf`-style arguments that should be type-checked against a format string). For more information on attributes for functions, see the gcc 3.4 documentation at:

<http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Function-Attributes.html#Function-Attributes>

Note It is recommended that you do not utilise the `__func__`, `__FUNCTION__`, `__PRETTY_FUNCTION__`, and all of the other variants of these constructs as they expose internal details of the code and result in security gaps.

The `__FUNCTION__ = sub` and `__PRETTY_FUNCTION__ = void a::sub(int)` identifiers are not preprocessor macros. In the GCC 3.3 compiler and earlier versions, in C only, `__FUNCTION__` and `__PRETTY_FUNCTION__` were treated as string literals. `__FUNCTION__` and `__PRETTY_FUNCTION__` were used to initialize character arrays, and concatenated with other string literals as required. The GCC 3.4 compiler and later versions treat `__FUNCTION__` and `__PRETTY_FUNCTION__` variables in the same way as `__func__`. In C++, `__FUNCTION__` and `__PRETTY_FUNCTION__` are variables.

Prototyping Functions That Take a Format String and Arguments

Format errors - mismatches between the format specifiers in a format string and the arguments passed to be formatted there - can cause bugs ranging from garbled output to invalid pointer de-references (which can crash many routers). So it is very important to keep format arguments aligned with the format string.

The gcc compiler can validate variable format arguments against the corresponding format specifiers in the format string, for every function that it knows about that takes a format string and arguments. Cisco mandates format checking where possible; therefore Cisco-defined functions that need format checking must be tagged for compiler format checking. This is done with the GNU directive `__attribute__`, applied to the function prototype. For example, the prototype for `buginf_noscan()` in `cisco.comp/cisco/include/cisco_io.h` is:

```
extern void buginf_noscan(int msg_level, const char *fmt, ...)
    __attribute__ ((format ( __printf__, 2, 3)));
```

The Cisco IOS code base already contains over 500 functions that accept a format string; and new ones are being added at a rate of several per month. Any time you create a function that takes a format string, you need to add this directive to its prototype, so that the compiler can help prevent formatting errors. The directive is placed in the prototype between the closing parenthesis of the argument list and the following semicolon, and takes the above syntax. The two numbers are the argument index for the format string, and the argument index for the first variable argument to be formatted into that string. In the example above, the “2” says that the format string is the second argument to the function, and the “3” says that the variable args (“...”) start immediately after the format string, at argument 3. You can find lots of examples with `gid` or `escape` - just search for `__attribute__` (note the *double underscore* both before and after).

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.

Note One exception to using typecasting is the following explicit cast, which is identical in both C/C++ - `sizeof((char) '\0')` - and the result is 1. This explicit cast would resolve the problem of having two different outcomes of `sizeof('0')` with different languages, where, for example, in C++, the result is 4 and in C, the result is 1.

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. In particular, implicit conversions (such as in `varargs` lists) will promote a `uchar` or `ushort` to `int` (since there will be no loss of precision). For instance, this means that a `uint8` will need to be formatted as `%d` and not as `%u`.

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 680x0-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.

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.

Note When updating code, “white space only” changes are not accepted in code review because they very likely will cause sync conflicts when doing branch syncs, collapses, or merges. Code can only be reformatted when a change in nesting level (such as adding an enclosing “if” statement) necessitates the addition or deletion of indentation.

A.4.1 Specific Code Formatting Issues

The following points address specific code formatting issues:

Command Length

The maximum number of characters for a command is limited to 255.

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.

Column Width

All code must fit in 80 character space columns.

Standard Indentation

Standard indentation should be applied as follows:

- The TABS standard is equivalent to eight spaces.
- The indentation standard is four spaces.

The exception to this rule is `switch` statements, where the `case` statements are kept at the same indentation level as the enclosing `switch { ... }`.

Any number of tabs and spaces can be combined to yield the required amount of indentation, although it is certainly desirable that the total number of characters used be minimized (tabs first, then 8-n spaces.) Any reasonable editor and nearly all tools can do this correctly. Deal with it. The characters used for indentation should not be gratuitously changed without other edits in the same line.

The major tool that looks weird when tabs+spaces and spaces-only indentation are mixed in the same source file is the output from context diffs. When the indication characters are prepended to the lines (“!”, “+”, “-”), a line indented with a tabs first won’t actually move, while one with spaces will have the entire line shifted.”

Diffs generated for human consumption—that is, code review and not to be applied as patches—should be made with ‘-t’, which instructs diff to expand all tabs into the appropriate number of spaces so that the leading ‘!’, etc. will not affect the columnar output and hence readability.

Checking Indentation: diff_nits.pl Utility

An indentation tool is available. This utility checks your diffs to ensure that they follow the coding style standard. To use the utility, you must run it within a view and a VOB. The tool is available at the following location:

```
/auto/nsstg-tools/bin/diff_nits.pl
```

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.
 */
}
```

Curly Braces

Put the curly brace on the line following the function header. If you put the curly brace on the same line as the function arguments, it can cause problems (for example, `cscope` decides that everything after the function is a part of the function and it also breaks some Emacs commands). Here is an example of correct placement of curly braces:

```
void function_header (int arg)
{
    function();
}
```

The following example code illustrates correct usage of curly braces, spacing, and indents.

```
static boolean
example_function (type_1_t * arg1, type_2_t arg2)
{
    type_1_t * x;

    x = function_call(arg2);

    if (x == arg1) {
        while (x == arg1) {
            type_1_t * y;

            for (y = x; y != arg1; y = next_type_1(y)) {
                switch (number_from_type_1(y)) {
                    case 1:
                        do_something_with(1, arg2);
                        break;

                    case 2:
                        do_something_with(2, arg2);
                        break;

                    default:
                        do_something_else();
                        break;
                }
            }
        }
    }
    return (TRUE);
}
```

The above example follows standard coding conventions related to curly braces, spacing, and indentation as listed below:

- Provide one space between the function name and the parameter list.
- Place the return types of the functions in the preceding line.
- Provide one space after ‘if’, ‘while’, ‘for’, and other conditions.
- Case labels should not be indented.
- Use the standard indentation of 4 spaces in the editor whenever the indentation is used.
- Use the standard tab spacing of 8 spaces in the editor whenever the tabs is used.
- Enclose return values within parentheses.
- Remove spaces between the name of the called function and the argument list.
- Keep the opening curly braces in the same line for the ‘for’, ‘if’, ‘while’, etc, conditions. Provide a single space between the opening curly braces and the preceeding parenthesis for these conditions.

Keep the opening ‘{’ and the closing ‘}’ curly braces in appropriate pairs, irrespective of the requirement of the coding language.

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. */
}
```

Assignments in conditional statements are strongly discouraged (except where the flow-of-control would be demonstrably more complicated without them) and should not be used in simple situations like this:

```
if ((str = malloc(BUFSIZ)) == NULL) {
    return;
}
```

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);`

Spaces around Operators

Spaces are used on both sides of operators in Cisco IOS. For example, use `1 + 1` rather than `1+1`.

Return with Parentheses

The `return` statement can be used with parentheses around the variable.

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)
 */
#ifndef 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."  
 */
```

Switch Statements

Case statements are indented at the same level as the enclosing switch. Each case should end with break, continue, return, /*FALLTHRU*/, or in special circumstances, goto. Don't put a break after any of the others, because an unreachable statement will cause compile warnings or static analysis warnings. The /*FALLTHRU*/ comment allows checking by static analysis tools.

For example:

```
switch (number)  
{  
case 0:  
  
    printf("You entered zero. I assume you meant 1.\n");  
    /*FALLTHRU*/  
  
case 1:  
{  
    uint count;  
    frobozz(&count);  
    if (count > min_required)  
        retval = count;  
    break;  
}  
default:  
...  
}
```

In this switch statement:

- 1 The variable count is meant to have a very short lifetime, because it is effectively a temporary variable, and its scoping reflects that.
- 2 The break occurs within the scoping, as do all other statements for this case.

For more information about switch statements, see section A.6 “Coding for Reliability.”

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 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.

Note The convention in IOS is to *not* use mixed case names.

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.

Note Preprocessor symbols defined in header files included by more than one or two C files should have at least a two character prefix.

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.

Note The use of `externs` in source (`.c`) files will cause static analysis warnings, and will later need to be fixed.

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 defensively*. 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-NUL pointer unless your function is clearly commented as requiring a non-NUL pointer. This comment should appear in the header file and in the function's header comment.

Switch Statements and Default Cases

A switch statement for an integer type should always contain a default case. Do not allow an unhandled value to fall through into the code that follows the switch statement.

Using an enumerated type for switch statements should always be considered when all possible values can be listed. In this situation, you should carefully consider removing the default case. If any case values are not present, the compiler generates a warning. This helps keep code up to date when new enumeration constants are added to the enum type. Note, however, that if the switch value is not a legal enum value, none of the cases are executed, and no error is generated. Therefore, this should be done only when validity checking has already been done or is not needed.

In other words, a switch statement for an enumerated type should not contain a default unless it's impractical to list all possible values explicitly. What we're trying to emphasize here is that while an "int" discriminator has too many possible values to list them all, an enum discriminator is much more finite, and hence inherently more feasible to explicitly state all values. In a broader sense, the two have this basic difference between them, and hence they should be thought about in two different ways.

For example, `timer_type`, which is defined as:

```
enum
{
    TIMER_NONE,
    TIMER_AGER,           /* our timer is for aging out this entry */
    TIMER_WAIT,           /* our timer is waiting for a InARP */
} timer_type;
```

is used in a switch statement in the `atm_arpserv.c` file, as shown here:

Note The following is good only if we're confident that `entry->timer_type` will only be set to an enum value. A comment saying that it had been range-checked earlier in the code, or some other argument to rule out that possibility might be in order.

```

switch (entry->timer_type) {
    case TIMER_NONE:
        DEBUG_ATM_ARP(entry->idb->hwptr)
            buginf("\nARPSEVER (%s): unknown timer expiry (vc %d, %i)",
                   entry->idb->namestring, entry->vcid, entry->ipaddr);
        break;
    case TIMER_AGER:
        /*
         * If we have a call in place, we'll InArp to see if they're still
         * there. We time the entry out in any case. If they are still
         * there, their response will create a new entry.
         */
        DEBUG_ATM_ARP(entry->idb->hwptr)
            buginf("\nARPSEVER (%s): entry for %i timed out",
                   entry->idb->namestring, entry->ipaddr);

        if (entry->vcid != 0) {
            send_inarp(entry);
        } else {
            entry_destroy(entry);
        }
        break;
    }

    case TIMER_WAIT:
        /*
         * The cretin on the other side doesn't want to talk to us. So
         * we'll just take our connection and go home.
         */
        if (entry->retrans_count > INARP_MAX_RETRANS) {
            DEBUG_ATM_ARP(entry->idb->hwptr)
                buginf("\nARPSEVER (%s): No response on vc %d. "
                       "Hanging up.",
                       entry->idb->namestring, entry->vcid);

            hangup(entry->idb, entry->vcid, ATM_NORMAL_UNSPECIFIED);
            entry_destroy(entry);
            break;
        } else {
            /*
             * We will send out a retransmission. This is needed because
             * sometimes someone along the path drops our first one.
             */
            DEBUG_ATM_ARP(entry->idb->hwptr)
                buginf("\nARPSEVER (%s): vc %d wait timer expiry. "
                       "Retransmitting.",
                       entry->idb->namestring, entry->vcid);
            send_inarp(entry);
        }
    }
}

```

Notice that there is no default case—this is as it should be.

Note There should be a `break` here, even though it's implied by the end of the switch—someone might add another case later.

Note Finally, whenever we omit a default, we should put a comment saying the omission is intentional and why (for example, “to force attention here in case someone adds a new enum value”). It is wise to include a comment explaining why you’re confident that range checking isn’t required.

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.

Format Specifier for a Pointer

`%p` is the correct format specifier for a pointer.

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 freeing the same block. `validmem()` is an expensive 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.

enums

Avoid the use of int variables to pass to functions that take enums, or vice versa, or the assignment of enum variables to int variables or vice versa, otherwise unforeseen problems can occur because of typecasting. The key point is to pass enums as enums, without typecasts, either explicitly or implicitly to anything else. A good practice would be to typedef an enum and use that definition to declare all the parameters/variables that would be assigned the enum values. Although some compilers do not check mismatches of enum types in function parameters, distinct enum types should still be used there for clarity.

Furthermore, use static analysis to do enum type checking.

The following example of using enums comes from `ios_macros.h`:

An `.enum` facility is fully documented in the `enum_define` script, which resides in 12.2S and 12.4T at `/vob/ios/sys/scripts/enum_define` or `/vob/cisco.comp/make-lib/scripts/enum_define` depending on whether the script has been Singlesourced or not.

```
/*
 * The following is a lightweight technique that allows you to keep enums
 * and their corresponding string definitions in sync. To use, first create
 * your map of enums to strings:
 *
 * // foo.h
 *
 * #define MY_ENUMS(list_macro) \
 *     list_macro(MY_RED,      "red"), \
 *     list_macro(MY_GREEN,    "green"), \
 *     list_macro(MY_BLUE,     "blue"), \
 *
 * enum my_enum
 * {
 *     MY_ENUMS( ENUM_MAP_DEFINE_ENUM )
 *     MY_ENUM_MAX
 * };
 *
 * This initial use of MY_ENUMS above, with ENUM_MAP_DEFINE_ENUM, will create
 * the list of enum definitions.
 *
```

```

* Now declare an array which will be used to house the corresponding string
* representations:
*
* // foo.c
*
* const char *const my_enum_strings[MY_ENUM_MAX] =
* {
*     MY_ENUMS(ENUM_MAP_DEFINE_STRING)
* };
*
* And here we use MY_ENUMS again, but with ENUM_MAP_DEFINE_STRING this time,
* to populate my_enum_strings with the string representation of our enums.
*
* e.g. my_enum_strings[MY_RED] is now "red"
*
* For an alternative approach, check out /vob/ios/sys/scripts/enum_define.
*/
#define ENUM_MAP_DEFINE_ENUM(enum_arg, string_arg)      enum_arg
#define ENUM_MAP_DEFINE_STRING(enum_arg, string_arg)    string_arg

/*
* An extension to this technique provides the same simple robustness
* for bitfield enums. These are enums where typically you would have
* a contiguously increasing series of bit sequence values e.g.:
*
*     enum my_enum_bitvalues
* {
*         MY_RED    = 1 << 1,
*         MY_GREEN = 1 << 2,
*         MY_BLUE   = 1 << 3,
*         // etc...
*     };
*
* Changes to this enum list are prone to typos and gaps in the bit sequence.
* These make it unreliable to perform string lookup, assuming a contiguous
* bitspace. However, using list macros we can enforce a strict mapping e.g.:
*
*     // foo.h
*
*     #define MY_ENUMS(list_macro)          \
*         list_macro(MY_RED,      "red"),    \
*         list_macro(MY_GREEN,    "green"),  \
*         list_macro(MY_BLUE,     "blue")
*
*     enum my_enum_bitshifts
* {
*         MY_ENUMS(ENUM_MAP_DEFINE_ENUM_BITSHIFT),
*     };
*
*     enum my_enum_bitvalues
* {
*         MY_ENUMS(ENUM_MAP_DEFINE_ENUM_BITVALUE),
*     };
*
* which creates:
*
*     enum my_enum_bitshifts
* {
*         MY_RED_BITSHIFT_VALUE,

```

```

*
*      MY_GREEN_BITSHIFT_VALUE,
*      MY_BLUE_BITSHIFT_VALUE,
*  } ;
*
*  enum my_enum_bitvalues
* {
*      MY_RED    = 1 << MY_RED_BITSHIFT_VALUE,
*      MY_GREEN = 1 << MY_GREEN_BITSHIFT_VALUE,
*      MY_BLUE   = 1 << MY_BLUE_BITSHIFT_VALUE,
*  } ;
*
* Now we can perform string lookups based on the bit position, as we are
* guaranteed that each enum occupies one slot in a contiguous bitspace e.g.:
*
* // foo.c
*
* const char *my_enum_bitvalue_to_string (const unsigned long value)
{
    const char *const my_enum_strings[MY_ENUM_MAX] =
{
    MY_ENUMS(ENUM_MAP_DEFINE_STRING)
};
size_t index;
*
if (!value) {
    return (common_str_unknown);
}
*
index = LOW_BIT_IN_LONG(value);*
if (index >= ARRAY_SIZE(my_enum_strings)) {
    return (common_str_unknown);
}
*
return (my_enum_strings[index]);
}
*
* my_enum_bitvalue_to_string(MY_RED) is now "red"
*/
#define ENUM_MAP_DEFINE_ENUM_BITSHIFT(enum_arg, string_arg) \
enum_arg ## _BITSHIFT_VALUE

#define ENUM_MAP_DEFINE_ENUM_BITVALUE(enum_arg, string_arg) \
enum_arg = 1 << enum_arg ## _BITSHIFT_VALUE

/*
* The following are provided to allow clients to provide their own
* suffixes for bit values:
*/
#define ENUM_MAP_DEFINE_ENUM_BITSHIFT_SUFFIX(enum_arg, suffix, string_arg) \
enum_arg ## suffix

#define ENUM_MAP_DEFINE_ENUM_BITVALUE_SUFFIX(enum_arg, suffix, string_arg) \
enum_arg ## suffix = 1 << enum_arg
/*
* And these are for where the bitvalue has a suffix, but the sequential
* enum does not and you need to share the same macro definition, hence
* the unused suffix argument. Best really just to use the defaults and
* change your code, but if that's not possible then do this.

```

```

*/
#define ENUM_MAP_DEFINE_STRING_NOSUFFIX(enum_arg, suffix, string_arg) \
    ENUM_MAP_DEFINE_STRING(enum_arg, string_arg)

#define ENUM_MAP_DEFINE_ENUM_BITSHIFT_NOSUFFIX(enum_arg, suffix, string_arg) \
\
enum_arg

#define ENUM_MAP_DEFINE_ENUM_BITVALUE_NOSUFFIX(enum_arg, suffix, string_arg) \
\
enum_arg = 1 << enum_arg

```

A.7 Coding for Usability

When writing code, you should be conscientious of how it will interact with the end user, and how the user will perceive it. For example, the following considerations should be kept in mind:

- Users have widely varying degrees of technical competence.
- Users typically won't interact with our boxes unless installing and configuring them, or else troubleshooting an issue- all can be highly stressful situations, so we should do our best not to exacerbate the circumstances.
- Users don't have access to source code, and might not even have ready access to whatever standards documents relate to their interaction with the router.
- Users similarly don't have the benefit of Cisco culture, and may have already been damaged by some other vendor's unenlightened way of doing things.
- Users might be in a less than optimal situation, trying to make the best of a dire situation, and will see the world from a very different perspective than developers in their offices.

What does this mean? Well, the implications are endless, but here is an example where several possible disconnects could be avoided: the user could be stuck debugging a router in an unheated shack 50 miles from civilization, with a 24-line terminal, 9600, and no scrollback. In this case, the user first needs to be able to limit debugging to useful information so that the useful information doesn't get pushed off the screen by something entirely useless. And second, if the output is too voluminous, either the box may crash before the useful information ever gets displayed or else it may simply be lost in the noise. This embodies the last example.

Not having source code means that a lot of things that we take for granted (because we can easily verify by a quick glance at the source) aren't available to users or to the TAC. For instance, printing connection IDs as %x is a *really bad idea*. Why? Because for a value such as 1234, it is:

- 1 Ambiguous whether this is hex or decimal.
- 2 Possible that values are displayed inconsistently in hex and decimal, depending on the particular place in the source.
- 3 Counter to the radix that is used either in a standards document or in another piece of equipment, such as a packet sniffer.

The user might see **port 19** in his debugs and not understand why his trusty packet sniffer hasn't been able to capture a single datagram. Why indeed? Because the debugs are using hex, even though the standard and the sniffer both use decimal. Ouch. This is not a far-fetched example; it has happened numerous times.

Always display the radix with a prefix (e.g. “0x” for hex) or with information in a heading for tabular data, unless the standard notation for the information is blatantly obvious (e.g. port numbers in IPX are always displayed as 4 hex digits). Rarely are things that obvious, though.

Another, more subtle, implication of not having access to the source is much more insidious. There might be places where a connection ID is displayed in debugs as:

```
conn_id = %x  
conn_id = %d  
conn-id = %d  
conn-id=%d
```

And so forth. The user might have turned on too many debugs, or the debugs might simply not have enough granularity, and the user (or TAC) is left to sift through megabytes of output. If they are clever, it could bite them on the posterior because the user can write a script to `grep` out all debugs that contain `conn_id = 211` and miss the ones that contain `conn_id = 529`, or `conn-id = 529`, or `conn-id=529`, and so forth.

So, be consistent! Don’t assume that the first round of data reduction is going to be done by eye. It could be done by a machine that doesn’t equate `conn-id = 529` with `conn_id = 529`.

Use a standard prefix string for debug messages, especially one that is defined as a macro, to help enforce consistency. For example:

```
#define CT_DBG_PREFIX "\nCT: hndl=%lu: "  
  
...  
buginf(CT_DBG_PREFIX "incoming call-type=%d", new_node->call_id,  
new_node->call_type);
```

This also has the benefit of reducing subsequent edits to just one if the format of the prefix needs to be globally changed (as might happen if handles are scaled up from 16-bits to 32-bits, or from pointers to cookies).

Note There is no comma between `CT_DBG_PREFIX` and the format string because the compiler will perform string fragment concatenation.

A.8 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

Note Super-efficient code will do you no good if it isn’t above all else reliable. Remember, *a crashed router passes no packets, quickly or otherwise.*

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.

SES has an IOS performance analysis webpage at the following URL:

http://wwwin-ses.cisco.com/performance/IOS_PerformanceAnalysis.html

A.8.1 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, Ronald L. Rivest, and Clifford Stein, MIT Press, ISBN 0-262-03293-7

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.8.2 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()`—Do not use `sprintf()`, use `snprintf()` instead. Try to do as much formatting in one call as possible, rather than spreading your string formatting out over many small calls to `snprintf()`.
- `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.8.3 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.8.3.1 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 680x0 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.8.3.2 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 680x0, 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.

A.9 Coding for Scalability

The following sections give advice on coding for scalability. The information is taken from ENG-121114, “*Guidelines for Writing Scalable IOS Code*.”

A.9.1 Designing, Writing or Modifying IOS Code

When designing, writing, or modifying IOS code, please keep the following in mind:

- Assume that there could be more than 100,000 swidbs on the system. Therefore, never use a `FOR_ALL_SWIDBS` loop in your feature code, looking for swidbs that use your feature. Either keep your own list of IDBs or subblocks for your feature or (better yet) use the subblock API’s features. Also assume that other quantities are large, such as the number of locally configured IP addresses.

Avoid `FOR_ALL_HWIDBS` loops as well. (There is an exception to the injunction against use of `FOR_ALL_SWIDBS`: they are permitted in “init” code to initialize “primordial” swidbs for your feature or protocol.) However, don’t loop through all the interfaces on a fully configured box because you may be looping through thousands of interfaces, which would very likely trigger a Watchdog timeout or something equally bad.
- Do not add fields to the swidb. Instead, use the subblocking feature. A common misconception is that if your field takes only 1 byte, then it is a big waste of space to use a subblock for it. The flaw in this logic is that Cisco routers have hundreds of features, and only a relatively small subset is ever used on any given installation. It is tempting, especially when writing a new feature that has scalability goals, to optimize your implementation assuming that it is always required, and eliminate the additional memory for the subblock header. But remember that this is at the expense of all other features, and unless it is truly the case that your new feature will *always* be used, subblocking is best. IOS might benefit from subblocking even IP, because of systems with heavy use of Layer 2 tunneling protocols or IPv6.
- Remember that a lot of IOS code was written when it was reasonable to assume that the total number of interfaces was small. Thirty was a large number for most of the early IOS code. Therefore, when looking at existing code as a model for your own code (a good practice), be sure to check the existing code for scalability problems, and do not propagate any such problems to your code.
- Ask yourself, “Will this new feature need to be used on a scalable interface type?” If so, follow the advice in the next section. This is not always an easy question to answer, and it can significantly increase the amount of effort required to implement the feature. Get input from Marketing. If you’re still not sure, and decide to implement the feature in an unscalable way, still think about scalability in your implementation, and leave hints in comments about possible problems and any ideas you have about how they could be resolved if necessary.

A.9.1.1 Writing Code for a Scalable Interface Type

When writing code for a scalable interface type or for a feature that might need to be used on a scalable interface type, keep the following in mind. Many of these are general principles that we should all be familiar with already, but a look at the IOS shows evidence that we do need reminders.

- Know the order of magnitude of your algorithms.

$O(N^{**}2)$	bubble sort
$O(N \log N)$	qsort
$O(N)$	linear search
$O(\log N)$	tree or radix search
$O(D)$	hash search, where average bucket depth is D . Since D tends to grow with N when the hash space is too small, this is essentially $O(N)$ —so choose hash space wisely. Ideally, D is very small, yielding $O(1)$.
$O(N)$	unlinking an arbitrary element from a singly linked list
$O(1)$	unlinking an arbitrary element from a doubly linked list

This means that if you need to locate your scalable element, don't use a linear search! Subblocking alone, using the subblock iterators, will not make your code scalable. You must also provide a good access mechanism. Consider using `os/string_db.h`, which applies to octet strings as well as to character strings. If this does not seem to apply, then provide an access mechanism that is at worst $O(\log N)$. If you use a linear access mechanism anywhere, make it clear why a linear algorithm is acceptable in the context, and warn against use of the access method in other code that might run $O(N)$.

- Keep in mind when designing your algorithms that are iterated within some larger context. For example, if adding a single new interface is $O(N)$, then configuring the system will be $O(N^{**}2)$.
- Keep in mind that there are different N s to watch when designing your algorithms. Examples:
 - the number of hwidbs
 - the number of swidbs
 - the number of swidbs on a given hwidb
 - the number of a given kind of interface or subinterface
 - the number of local addresses
 - the number of interfaces for which a feature is configured
 - the number of packets processed
- Do not attach your feature to the hwidb. Currently, the number of hwidbs does not scale well in IOS, and so your feature won't scale well either. Various features that are currently attached to the hwidb are moving to the swidb to support greater scalability. But don't think that this means that you don't need to design for hwidb scalability. There are features that currently require hwidbs and are configured to many thousands.
- Minimize the number of dependencies your code has on the location that your feature is attached to. If your code runs on a subblock and has minimal dependencies on the subblock's parent, you can easily move it to a different context or support it under different contexts.
- Spend less time optimizing code and more time optimizing algorithms. IOS is full of code that is peephole-optimized to the n th degree, but runs in a context where it is executed millions of times, when a little extra thought could have produced code that would run only thousands of times. Optimize code only after you've optimized the algorithms and then determined that the code is in an inner loop that needs to run extremely fast. Most code in IOS is not packet-forwarding code, and the same tradeoffs do not apply.

- Profile your subsystems under a number of different configurations to find out where IOS is spending time. Find ways to measure scalability (space consumed, time spent configuring, CPU time used at runtime) and see how these measurements change with the number of interfaces that are in use. Measure before and after changes that you expect to improve scalability.
- Remember that for scalable features, it may be almost as important to optimize configuration-time algorithms as it is to forward packets efficiently. Reducing reload time makes a product more marketable, makes it appear more robust in demonstrations, and, most important, reduces down time for scheduled maintenance or unplanned outages (hardware failures, software failures, power failures). Reducing reboot time can dramatically improve MTTR (Mean Time To Repair), which is a key factor in determining availability for High Availability analyses. This is significant for our customers and must be taken seriously by development engineers!

A.10 Coding for Security

See section 30.1 “Secure Coding” for detailed information on this topic.

A.11 Coding for Multiple Operating Environments

In the last few years (late 1990s/early 2000s), IOS code (and runtime environment) have begun to be used as application code on other operating systems. For this reason, we’ve established some guidelines about crafting your code for maximum maintainability, safety, and performance in multiple underlying operating systems.

This section provides some basic information, such as definitions (operating system boundaries, modularity domains) and some background information (brief overview of header file usage).

A.11.1 Definition of “Operating System Boundary” in the IOS Code Base

Different operating systems have different boundaries. Most folks are familiar with the Un*x family of operating systems (OSs) and the boundaries those OSs form. But IOS has very soft, ill-defined boundaries and so presents some challenges when attempting to replace bits of it with some other operating system.

Another challenge in defining an “operating system boundary” is the fact that IOS network-application code assumes a very particular “runtime environment,” or RTE. The IOS RTE should be maintained (or emulated) across multiple OSs. In an ideal world, network-application code would never be exposed to the operating system, and it would only be exposed to the RTE for which it was coded.

Things aren’t so ideal, though, for a variety of reasons. One key issue is that there are operating-system leaks into the C header files that provide the RTE for the application code. Some of these leaks are just tradition carried on by operating-system developers (including IOS). And some are for performance reasons—that is, the implementation gets inlined into the include C module. And some are there because the API itself leans on macro definitions, inline functions, type definitions, etc.

So for the purposes of the following discussion, we can define *operating system boundary* as the code API and implementation that might change when the underlying operating system code source or version changes *and* that must be visible to the application layer. IOS code and developers shouldn’t care about the internals of the operating system being used—just its boundary code that must be exposed to the application.

A.11.2 Runtime Environment and IOS Code

This section does not address the question of how to run IOS code in multiple RTEs—for example, different thread/task models, scheduling models, etc. With few exceptions, IOS application code is written to a particular RTE (the IOS RTE), and the question of how to make that code portable to other RTEs is well beyond the scope of what is presented here.

A.11.3 Writing IOS Code for Use with Multiple Underlying OSs

Most IOS code (feature or driver code) should be runnable with little or no change when being compiled in the context of any of the supported underlying OSs. Sometimes, of course, some things really need to change.

The sections below cover the following topics:

- Issues with reusing source code across multiple underlying OSs
- Motivation for creating this set of guidelines
- The guidelines themselves
- Examples of use

A.11.4 Issues with Using Source Code Across Multiple OSs

The following subsections discuss three issues:

- Dependencies—Does the source code need to be changed to suit different OSs?
- Header files—How are their include relationships organized?
- Compilation—Is compilation conditional on any OS-specific macros?

How these issues are treated by you, the coder, determines how robust the code's behavior will be in the face of changes to the OS and RTE. The next few sections detail some of the issues; the sections following those provide some guidelines about solving the issues.

- Dependency Management
- Header File Relationships
- What Is a Modularity Domain?
- Implementation Incompatibilities Between OSs
- Meanings and Usage of Conditional Compile Macros

A.11.4.1 Dependency Management

The first thing to discuss is what's called a “dependency relationship,” which sometimes gets referred to as just “dependency.”

Most abstractly, a code unit is said to have a *dependency* on a second code unit when it requires a definition provided by the second code unit. In other words, a code unit cannot be complete without having all of its requirements met externally by other code units (and internally as well). The reason for using such abstract terms is that there are different types of dependency relationships a developer will see:

- Compile-time
- Link-time

- Runtime

Compile-Time Dependencies

You have a compile-time dependency when a code unit must reference a definition from a second code unit in order to compile successfully. In the C language, when a source file includes another source file for a struct/typedef definition or global variable/function declaration, that source file can be said to have a compile-time dependency on the other source file. If the definitions for the required dependencies are incomplete or inconsistent, then the compile will fail.

Link-Time Dependencies

You have a link-time dependency when a code unit must reference a symbol definition from a second code unit in order to link successfully. For a program to be executable, all symbol references must be fulfilled; if they are not, the link will fail. A link can occur statically or (in more recent times) dynamically. Note that all code units may have compiled successfully, yet when linked together, the link may still fail because some symbol definitions are missing or inconsistent, for example, incompatible type or storage class.

Runtime Dependencies

A runtime dependency is often referred to as a *semantic dependency*. A program may have been able to compile and link, but that doesn't mean it will run correctly. For example, a code unit may expect messages to be sent in a certain order by a second code unit; if that order is violated, then the program fails.

A.11.4.2 Header File Relationships

C header files define a shared API, occasionally with an implementation supplied via macros or inline functions. Header files do not, by themselves, define the scope of their usage. Usage patterns of header files cover the spectrum. Header files can be included by just a couple of source (.c) modules, or perhaps by nearly all source modules. Header file usage patterns do have at least two broad categories that deserve mention:

- Private—The header file is used only by source modules that all belong to the same “component” (for example, SingleSource component, IOS non-component-based subsystem, etc.).

A SingleSource component exists in its own repository with its own branching model. It has a well-defined public API, which resides under the `include` directory of that component. It contains private data (source code, private headers), which resides under the `src` directory of the component. It contains binary objects that are built under the `objects` directory. Finally, it manages a list of dependencies upon the API versions of other components. This means it should depend only upon the APIs of other SingleSource components. A SingleSource component is “owned” by the component owner—its feature development, release model, and branching requirement—that is integrated into a release train (integration branch) via an activity called “publication.”

- Public—The header file is used by source modules belonging to different components (for example, different SingleSource components, or different non-component-based subsystems, etc.).

Many source modules may include the same header file. This is an important point. In the C language, a source module may be a .c file or a header file. When header B includes header A, then any file that includes header B can be said to have a *transitive* dependency upon header A. In such cases, we say that a file *indirectly* includes header A.

It's fairly straightforward to determine the status of a header file in IOS:

- Step 1** Note all source modules that directly or indirectly include a specific header file. Do this across all images on all platforms, for each operating environment (IOS, ION/modular IOS, etc.).
- Step 2** Calculate for each image the set of modularity domains that are associated with those source modules.
- Step 3** See if any image requires a subset of the noted modularity domains (see section A.11.4.3, “What Is a Modularity Domain?”). If that subset is neither the empty set nor the full original set of modularity domains, then the header file is public. Otherwise, the header file is private.

A.11.4.3 What Is a Modularity Domain?

A *modularity domain* is some notional set of *items* that cleave together in some way that is useful for creating code that runs on the router. Those items can belong only in a single such modularity domain. For instance, in IOS there are SingleSource components. Each SingleSource component is associated with all source modules belonging to that component. So the component is the SingleSource modularity domain for those source modules. Similarly, the modularity domain of non-SingleSource IOS source modules is the `makesubsys`-style subsystem, of which they are a part.

Figuring out public versus private usage status of every header file in IOS is very tedious and not very connected to everyday programming tasks; SCM (Source Control Management) projects such as SingleSource make it easier to calculate the status (and enforce same) of the public/private usage of header files. Also, over time, a header file might change its status from public to private (or vice versa) because of different usage patterns. Whether or not such migration is desirable is a different matter (see section A.11.4.2, “Header File Relationships,” for guidelines on keeping private header files private).

Other types of analysis can be done with the boundaries of modularity domains. For instance, whenever a source module or header from modularity domain A includes a header from modularity domain B, we say that modularity domain A has a *compile-time* dependency on modularity domain B.

Link-time analysis of modularity domains can also be done. For instance, an IOS implementation of `sub_foo.o` might use `sub_bar.o` via symbols `bar_a` & `bar_b`. We say that modularity domain `sub_a` has a link-time dependency on modularity domain `sub_b`.

Conditional Inclusions and Definitions

Conditional inclusions can occur explicitly or implicitly. When include statements are surrounded by a preprocessor-based `if-macro` statement, then we have an *explicit* conditional inclusion. The `if-macro` that's used to trigger the conditional inclusion is passed in from the environment or from the compiler command line. When include statements use a *search path* specified by the environment or compiler command line, then we have an *implicit* conditional inclusion.

Similar to conditional inclusions, a conditionally defined macro may have multiple definitions, or different macros may be defined depending on the compilation context. More generally, conditional inclusions and definitions result in what is called *conditional compilation*.

Combinatorics of Conditional Compilation

One consequence of using conditional inclusions or definitions is that, depending on the compilation context, a generated object file can have different binary implementations. For every variant of conditional inclusion or definition, slightly (or even largely) different binary code will be generated. Here's a recipe for determining the number of binary implementations:

- For a given .c file and the entire include graph (direct and transitive) of that .c file, enumerate the set of unique if-macros as well as the set of unique search paths, compiler command-line parameters, and environment settings.
- Place conditionals that are intrinsically mutually exclusive of one another in the same groups.
- Sum the items in each group and multiply numbers together. This gives you the maximal number of binary implementations a given source code module can generate. However, the actual number is likely be less as some conditionals are never used in conjunction with other conditionals.

For example, given a source code unit (.c file and include graph), we find these conditionals: OBJECT_4K, OBJECT_860, OBJECT_68, ios build, neutrino build, 32-counters, and 64-bit counters. These form three groups: CPU(OBJECT_4K, OBJECT_860, OBJECT_68), OS(ios build, neutrino build), and COUNTER(32, 64). Thus, there will be $3 * 2 * 2 = 12$ maximum possible binary implementations of that code unit.

Note Because we know that there are no Neutrino builds for m68k CPUs, the actual number will be smaller.

The point of this exercise is to show that using conditionals has a price, that is: the more conditionals, the more objects generated.

Maintainability of Conditionally Compiled Code

Another consequence of conditional compilation is that code usually becomes less maintainable. Here's a nonexhaustive list of reasons why:

- Combinatorics—As noted in the previous section, simply keeping track of the variations of what is produced gets cumbersome and consumes time and resources.
- Where's the difference?—Most examples of conditional compilation cause a difference in very little of the resulting object file. Yet it's not usually very easy to identify those differences quickly or accurately by skimming through the source code. So bugs can often be overlooked (or hard to find) compared to other ways of organizing the source code.
- Readability—Conditional-compilation directives are aesthetically ugly and greatly reduce readability of the code if used pervasively.
- What's another macro?—Human psychology is such that people naturally maintain the *style* of a source module when modifying it. Usually. Use of macros is no different: a source module littered with macros is going to attract more macros as time goes by. This does not encourage good coding practice.

Transitive Header Inclusion and Conditional Compilation

Compile-time dependencies are transitive. So if header A includes header B, then all source code that includes header A also indirectly includes header B. If header B contains conditional compilation directives, then all source that includes header A is subject to those conditional compilations.

This dependency has unfortunate combinatoric consequences if header B is a public header file: all source modules that depend on B now produce different object file content, depending on the value of the preprocessor flags used in the conditional compilation directives. See the above section “Combinatorics of Conditional Compilation,” for how this affects image builds.

If the public header files contain conditional compilation directives (other than CPU and OS directives, which are accepted as necessary evils), there’s not much that can be done to limit the combinatoric damage done to the build system.

A.11.4.4 Implementation Incompatibilities Between OSs

Different OS header files have different ways of presenting or implementing APIs. A simple example exists with `<ctype.h>`. Both IOS and QNX operating systems define an implementation inside the `<ctype.h>` header file. Unfortunately, the implementations are different enough that including the wrong one results in link-time or runtime difficulties.

For example, the `isalpha()` function is implemented using the `_IsTable` array in QNX, but is implemented using the `_ctype_` array in IOS. As a result, code compiled against the IOS `<ctype.h>` fails to link against the QNX `libc.so` shared object. See section A.11.6.1, “Guidelines for Application-Code Developers,” for guidance in resolving this problem.

A.11.4.5 Meanings and Usage of Conditional Compile Macros

Sometimes people invent broad, generic macro definitions and hook a variety of compile-time conditional behaviors to them. For example, the ION project used `IOS_ON_NEUTRINO` for anything that behaved differently in ION. Such hooks almost invariably turn out to be bad ideas. Here’s why:

- 1 It’s too easy to use the hook instead of thinking about a more modular way to fix the problem.
- 2 Mission-creep sets in: what the conditional is used for eventually broadens to the point where it doesn’t really mean anything in particular. Then points 3 and 4 make an appearance.
- 3 Many uses of the hook turn out to be wrong: the user meant to indicate some other, more specific condition (like OS-type).
- 4 The code behavior breaks when a new conditional “choice” appears in the code base. This is a bit more subtle kind of breakage: the meaning that the coder *wanted* the hook to have remains the same, but the *actual* meaning has changed. For instance, suppose some source code has a conditional if-then-else ladder based on OS-type and covers all OS-types exhaustively. When a new OS-type is introduced, that source code is broken unless someone edits it to add a new if-then-else case for the OS-type.

See section A.11.9, “Task-Oriented Coding,” for how to write code that avoids these problems.

A.11.5 Motivation for Creating This Set of Guidelines

There are two, apparently opposing, movements occurring within the core IOS technology framework, which must be recognized and understood by IOS code developers.

The first is the push to modularize software into “components.” Each component provides a clearly defined set of interfaces (APIs); understands which other components it depends on for services; and is decoupled from specific development trains. See the EDCS documents 133327: “*Component Directory Structure*” and 158227: “*IOS Modularity Guidelines*” for details.

The second movement is to run IOS code on top of “foreign” OSs. There are various motivations for this. One is to make it easy to port software written for other operating systems. Another is to provide separate process address spaces, which enables fault containment and process restart. See the EDCS document 156882: “Cisco ION Architecture.”

The component model has been reconfigured slightly to cohabit with various OSs, which might include QNX Neutrino, Linux, VxWorks and, of course, IOS. The component machinery will manage a hierarchy of target objects, which will account for the OS as well as the CPU. And a limited number of OS-dependent conditional definitions will be allowed in public header files, as well as CPU-dependent conditionals.

The issues outlined in the section A.11.4, “Issues with Using Source Code Across Multiple OSs,” were seen when a new underlying operating-system (QNX Neutrino) was being introduced into the IOS code base. Many other issues were seen and dealt with on a “business as usual” basis, but the above issues were identified as needing to be addressed via coding guidelines.

A.11.6 Guidelines for Writing IOS Code That Runs Over Multiple OSs

The following sections outline current best practice for how to write IOS code for maximum safety, maintainability, and performance when the underlying OS may vary.

A.11.6.1 Guidelines for Application-Code Developers

The following sections provide guidelines for application-code developers:

- Treat OS-Environment Build Targets Like CPU-Build Targets
- Avoid References to OS Macros in Public Headers and Source Files
- Do Not Create Your Own Conditional Macros in Public Header Files
- Code for Maximum Maintainability Whenever Possible
- Code Differences in Publicly Defined Structure Field Definitions
- Coding for Performance

A.11.6.1.1 Treat OS-Environment Build Targets Like CPU-Build Targets

The underlying OS is targeted in the build in the same way as the CPU is targeted. That is, application code includes header files that may then include CPU-specific header files. But this inclusion mechanism is OS-independent in terms of the header file names that the application code uses.

For example, take `errno.h`. To ensure that all objects compiled for, say, QNX Neutrino have a consistent view of error numbers, `errno.h` might defer error numbers to the Neutrino component. In contrast, for IOS it includes the IOS so-called `ansi` component:

```
// front & end material deleted for clarity...
/*
 * OS Specific Includes
 */
#ifndef __QNXNTO__
#include COMP_INC(nto, errno.h)
#else
#include COMP_INC(ansi, ios(errno.h))
#endif /* __QNXNTO__ */
// end example
```

As seen in this example, lack of a viable OS macro definition means that IOS is the underlying OS. The IOS-based OS may also be detected through the predefined `_IOS_` macro.

If the conditional directive involves including different header files for different kinds of underlying OSs, then wrap the choices in a OS-neutral header file, as shown in this example.

A.11.6.1.2 Avoid References to OS Macros in Public Headers and Source Files

It is legal to have conditionals that depend on the definition of OS macros in public header files and in private source files. But it is usually wise to avoid such entanglements, for reasons given earlier in this section. See section A.11.9, “Task-Oriented Coding,” for how to write code that behaves differently in different operating environments.

A.11.6.1.3 Do Not Create Your Own Conditional Macros in Public Header Files

Do not create conditional directives for use in public header files. As mentioned earlier in section “Transitive Header Inclusion and Conditional Compilation,” conditionals in public header files have effects on how and where object files are generated, and consequential effects on build time, disk space needed, directory structures, etc.

Having conditional directives in private headers and in .c files is an ugly but accepted practice. Remember that private code sometimes appears in the public interface at a later date and so may have to be reworked at that time.

A.11.6.1.4 Code for Maximum Maintainability Whenever Possible

Coding for maximum maintainability is a many-splendored thing. In the context of coding for multiple operating environments, it means:

- Coding in a “task-oriented” fashion. (See section A.11.9, “Task-Oriented Coding.”)
- Binding to the operating environment at some point (link-time, runtime, or compile-time) that results in the most code reuse and the least code duplication.

The programmer can choose to bind his/her code to a specific OS at link-time, runtime, or compile-time. Below is a discussion (with examples) of each of these choices. This list is ordered from most preferred to least preferred:

- Link-time binding—Code can be bound to a particular operating environment at link-time; different object files can be bound according to different OS/RTE requirements.
- Runtime binding—Code can probe the operating environment at runtime to discover what behavior a particular module should have. The binary code used to implement has the footprint of the sum of the range of operating environments, because the link dependencies pull in object code from the complete set of operating environments.
- Compile-time binding—Code can be bound to an operating environment using conditional preprocessor directives passed in via the command-line “pre-defined” macros. The resulting object code is specific to the set of predefined macros.

Follow this order of precedence if nothing else is known about the code in question. However, a more detailed set of guidelines follows.

If the code differences between operating environments are large enough for a function (and the code is not in the normal data path), then link-time binding is recommended. For example, the following code lives in module `a.c`:

```
#ifdef IOS_ON_NEUTRINO
/*
 * edt: oneshot_init_set
 *      marks 'resname' resource as having been initialized. This
 *      marking will persist until the operating system is rebooted.
 *      Whether or not a resource is marked as initialized can be
 *      queried via oneshot_init_get().
 */
int oneshot_init_set (const char* resname) {....};
int oneshot_init_get (const char* resname) {....};
#else
int oneshot_init_set (const char* resname) { return (1); }
int oneshot_init_get (const char* resname) { return (0); }

#endif
```

The `oneshot_init_*` functions could easily be split into two files, `a_iion.c` and `a_ios.c`.

If the code difference between operating environments is very small and is in the body of a function, do the following:

- Follow the “task-oriented programming” guidelines and ensure that the actual compile/link/run-time difference is in the initialization code. See section A.11.9, “Task-Oriented Coding,” for more information.
- Attempt to arrange the initialization code differences as link-time choices. If that doesn’t work, try a runtime probe. Use a compile-time choice only if the code is critical for router performance.

A.11.6.1.5 Code Differences in Publicly Defined Structure Field Definitions

Sometimes a code difference between operating environments touches the public-header definition of a structure. Such changes to public-header definitions are ugly, but accepted, especially if they relate directly to the underlying operating system being used, for example, if `__IOS__` is defined. But many times no such direct relationship exists. This section outlines guidelines to address the case where different operating environments suggest different field definitions in public structure definitions?

The trick is to notice that usually these *different* fields in the structure have little to do with the core usage of the structure. Instead, the fields usually have some more casual relationship to the core usage of the entire structure. So why is it tempting to place the fields in the existing structure? Nearly always, the reason is that doing so does not require any fancy lookup/navigation/init/fini code. The compiler takes care of allocating the field (it’s part of the structure) and takes care of access (a field is just an offset from the base “anchor” of the structure).

Note Adding miscellaneous fields into a public structure (because it makes structure allocation/lookup/finalization easier) has been a nuisance to the IOS code base from time immemorial: such “casual attachments” to the core public structures increase structure size for all platforms (even if they do not use the additional fields). They also increase the dependency footprint of all code that needs access to the core public structure.

Here are a few tips to arrange the public structure definitions to avoid such “casual attachments.”

- If the overall structure is always allocated from the heap, guard the allocation of the overall structure with a routine. Vary the behavior of that routine on a per-operating-environment basis. More specifically, have the implementation's new/init routine allocate and initialize memory blocks before the public structure, like this:

```
priv_ptr = malloc(sizeof(struct oe_privates)+ sizeof(struct
publics));
public_ptr = (unsigned char*)priv_ptr + sizeof(struct oe_privates);
```

To get to the private operating environment-specific block of memory, the code only needs to offset the base `public_ptr` by `-sizeof(oe_publics)`. Note that the coder needs to be cognizant of alignment requirements of both the public and private structures. A simple way to this need would be to have the private structure encompass the public one, like this:

```
struct oe_privates { struct oe_privates; struct publics; }
```

- If the public core structure is navigable via some key, then the conditionally included structure (or fields) can be broken out and looked up via the same key. This will result in slightly greater memory usage (a pointer per key per conditional structure), but it is useful in environments where the core public structure is not always allocated from the heap.
- Think of other techniques that take the conditionally included fields out of the public header files and into .c source modules.

The following code snippets show a data structure, defined in a `os-common.h` file, before and after reorganization.

Data Structure Before Reorganization

```
typedef struct thread_table_
{
    char *name;                                /* Name of this table */
    uint size;                                 /* Number of hash slots */
    thread_linkage **slots;                   /* Hash table pointer to slots */
    thread_linkage *head;                      /* Head of list */
    thread_linkage *tail;                      /* End of list */
    ...
    thread_delete delete;                     /* delete routine */
    #ifdef IOS_ON_NEUTRINO
        pthread_mutex_t table_lock;
    #endif
} thread_table;
```

Data Structure After Reorganization

```
typedef struct thread_table_
{
    char *name;                                /* Name of this table */
    uint size;                                 /* Number of hash slots */
    thread_linkage **slots;                   /* Hash table pointer to slots */
    thread_linkage *head;                      /* Head of list */
    thread_linkage *tail;                      /* End of list */
    ...
    thread_delete delete;                     /* delete routine */
}

typedef struct thread_table_ thread_table;
```

Data Structure in ion-only.h

```
#include <pthread.h>
#include <os-common.h>

typedef struct thread_table_posix_
{
    pthread_mutex_t tt_lock;      /* lock when modifying table */
    struct thread_table_ tt_table; /* actual table used by others */
} thread_table_posix_pre;
```

Data Structure in ion-only.c

```
thread_table *ipc_thread_table_init (uint table_size)
{
    thread_table_posix_pre *posix_pre_table;

    posix_pre_table =
        calloc_named(1,
                    sizeof(thread_table_posix_pre),
                    "IPC Thread");
    if (posix_pre_table == NULL)
        return NULL;
    ...
    pthread_mutex_init(&posix_pre_table->table_lock, NULL);
    return (&posix_pre_table.tt_table);
}
```

This example hides the OS-specific implementation of the table from the common code. Note that this code relies on the allocation of `thread_table` objects via `ipc_thread_table_init()` and on having all table manipulations guarded by functions that can be extended/provided in an OS-specific fashion. For instance, suppose the threads are added to the table using the `ipc_thread_append()` call. The `ipc_thread_append()` call might look like this:

```
void ipc_thread_append (thread_table *table, void *item)
{
    thread_linkage *element;
    leveltype status;

    element = (thread_linkage *)item;
    status = ipc_lock_table(table);
    ...
}
```

The call `ipc_lock_table()` is implemented differently in different operating environments.

A.11.6.1.6 Coding for Performance

Very occasionally, maintainability must be sacrificed for performance. In this case, the “compile-time” binding will be the only viable choice.

A.11.6.2 Specific Advice for Code in Source Components

This section includes special advice for code in source components.

Where to Home OS-Neutral Wrapper Headers

OS-neutral wrapper headers should be placed in a separate component from any headers supplied by a specific OS. The following (proposed) components are reserved for standard headers:

```
stdlib_wrap    ;; ONLY for ANSI/IOS standard headers (C89, C99,etc)  
pthread        ;; POSIX Pthread & Scheduling support headers
```

You may have a use that is common, but does not involve the standard headers included in the above components. If this is the case, then you may create a new component that contains the broadest possible range of headers that (a) usually change at the same rate and (b) are often included together by using code.

A.11.6.3 Specific Advice for Code in IOS sys/ tree

There is no sys/tree-specific advice.

A.11.6.4 Guidelines for Developers Dealing with OS Ports

Some of the work to integrate new underlying OSs (and OS versions) with IOS code involves the “wrapper” components discussed in the above section “Where to Home OS-Neutral Wrapper Headers.” It is important that *each* API that is new (or has changed in any way) is examined for semantic and syntactic compatibility with the other underlying OSs. Differences must be reconciled, either by merging in the headers or by wrapping the implementing functions.

A.11.7 Cross OS/Platform Driver Sharing

The Dev-Object Model is used for cross-OS and cross-platform driver sharing. Each dev object-based driver implements the dev object common function vector table, which allows the upper-layer code to do the device initialization, device enable, device destroy, and so on. Also, devices can provide additional functionality by publishing a device-specific (or device class) function vector table that gives the upper-layer code knobs to control that specific device (or class of device). For example, a SONET framer device would use a function vector table that allows you to set overhead bytes, set clocking source, collect statistics, and so on.

Another set of vectors is defined that allows devices to access OS-provided services, such as `printf()`, `malloc()`, `list()`, and `queue()` routines. These vectors are initialized once by the OS that supports dev objects, and then any dev object-based driver can access those services. These vectors come primarily from standard libraries.

Dev object common header files restrict the APIs available to dev object-based drivers to prevent access to APIs that are not supported. This restriction helps keep the infrastructure lightweight and makes it easier to sustain across multiple operating systems and branches. Support for dev objects is currently implemented in IOS, IOX, QNX, IOU, and diags operating environments. Dev object-based drivers have been implemented on platforms including 7300, 7600, 10000, and 12000.

Device drivers that are implemented using the Dev-Object Model can be shared across multiple platforms and operating systems. This model allows them to be reused completely and without modification for any platform or operating system specifics.

For more information on the Dev-Object Model, see section 6.16, “The Dev-Object Model.”

A.11.8 Techniques for Coding

Certain coding techniques are superior for producing code that is resilient in multiple operating environments and is most easily maintained over time. The following section outlines the attributes that IOS code should demonstrate and offer techniques to maximize those attributes:

- Coding for Maintainability and Safety

A.11.8.1 Coding for Maintainability and Safety

With the exception of data-plane packet-handling code (see the following section, “A Notable Exception: Data-Plane Performance”), IOS code should be written with maintainability and safety first and foremost. In some circles these two attributes are combined into one: robust coding. The term “robust” is certainly attractive, but a bit too vague for our purposes. So some definitions, of a sort:

- *Maintainable code*—Code that continues to compile and run correctly despite changes in module internals, using modules, used modules, and the operating environment.
- *Safe code*—Code that continues to run correctly in different RTEs and dynamically changing runtime conditions. Most code committed to the IOS source repository works on a good day in a fair wind. But code written with safety in mind responds to such variation far more consistently than code written for criteria such as performance, time-to-market, etc.

Coding for maintainability and safety involves some obvious techniques such as handling errors, using safe forms of library functions, keeping functions well-defined and focused, keeping algorithmic complexity to a minimum, etc. Coding for maintainability and safety in multiple operating environments means the following:

- Use “task-oriented” coding techniques. (See section A.11.9, “Task-Oriented Coding.”)
- Minimize assumptions about operating environment; make such assumptions explicit in the code (or at least in the comments).
- When the behavior of the code must vary with different operating environments, use link-time techniques whenever possible. Use runtime probe techniques sparingly, and avoid compile-time techniques (even in private component code) whenever possible.

A Notable Exception: Data-Plane Performance

The data-plane code has its own requirement that trumps the usual goals of maintenance and safety: namely, speed. In this case, link-time techniques are still preferred if the packet-forwarding rates are not impacted. However, runtime probes may not be practical, as they tend to require a global-var reference. In this case, compile-time techniques are acceptable as long as they do not occur in public header files.

A.11.9 Task-Oriented Coding

“Task-oriented coding” is a simple catch phrase indicating that the developer should code modules such that the runtime code is all about the task to be done, and not about the temporal or OS environment in which the code finds itself. That is, code the module so that platform and OS-centric material is restricted to the init/fini code for the module. Have the bulk of the code depend on the *task* conditionals that were set in the init-code (or by the compiler-defines if the compile-time method is used). That way, actual OS dependencies (and at least some platform dependencies) are isolated to a small segment of the code.

The following coding techniques can be used to make your code more task-oriented and so avoid unnecessary entanglements with OS (or other environmental) dependencies:

- Self-containment—Whenever possible, have your code reference its own data and not that of other modularity domains or surrounding environments. Coding this way insulates your code from incidental changes to code in other modularity domains.
- Code to features, not environments—Most feature code is about algorithms, protocols, and data flow. It is not about in which OS or RTE the code finds itself. So make sure the source code you write reflects this. If a particular OS environment dictates changes to the way your code behaves, code those changes and alternatives as alternative, operating environment-independent behaviors.
- Non-task-oriented code isolation—Sometimes your code must probe the OS at runtime. In nearly all cases, these probes can be done at init-time or fini-time. So try to place all probes of the environment in the init/fini code. In addition, place the init/fini code in its own source file. This increases self-containment and makes it easier to identify which parts of your program really are directly influenced by changes to the operating environment.

A.11.9.1 Example of Application Code Usage: Choice: Remote or Local Table Access

Often, runtime checks can be designed to choose code paths. For example, there are API functions that return whether the caller is running inside a classic monolithic IOS image or whether the caller is in a process separate from the Blob of unconverted IOS code. Most are implemented as inline functions that access process-global data (in the sense of a POSIX, not IOS, process), so that the cost of these calls is low.

Here's a small iprouting function that uses a fast runtime check to decide whether to make a (remote) registry call or find its IDB data in a local cache:

```
ushort iprib_idb2tableid (idbtype *idb)
{
    if (!idb)
        return (IPRIB_GLOBAL_TABLEID);

    if (process_running_blob()) {
        if (!idb->ip_info)
            return (IPRIB_GLOBAL_TABLEID);
        return (idb->ip_info->idb_rib_tableid);
    }
    return (reg_invoke_iprib_idb2tableid_remote(idb));
}
```

This strategy fails, of course, if `iprib_idb2tableid()` is running inside the Blob and the `iprib_idb2tableid_remote` service is moved outside the Blob! To make this code more resilient, initialization code should set a feature-specific flag (`iprib_service_remotely`) that indicates whether the caller and service provider are remote:

```
ushort iprib_idb2tableid (idbtype *idb)
{
    ushort table_id = IPRIB_GLOBAL_TABLEID;

    if (idb) {
        if (iprib_service_remotely) {
            table_id = reg_invoke_iprib_idb2tableid_remote(idb);
        } else if (idb->ip_info) {
            table_id = idb->ip_info->idb_rib_tableid;
        }
    }

    return (table_id);
}
```

The `iprib_service_remotely` flag should be set by an initialization routine, located in some other module that will be linked conditionally, depending on the operating environment:

```
# ION version
void iprib_probe_service_locality (void)
{
    /* the Blob has local access; everybody else must go fish */
    iprib_service_remotely = process_running_blob() ? FALSE : TRUE;
}

# IOS monolithic version
void iprib_probe_service_locality (void)
{
    iprib_service_remotely = FALSE;
}
```

A.12 Using Open Source Code in IOS Software

The Open Source Review Board (OSRB) team is an official team in Cisco that is responsible for setting Cisco's corporate policy with regard to open source software. The OSRB answers questions and formulates and refines policy as required. The membership consists of qualified engineers, attorneys, and business persons from organizations throughout Cisco.

For Cisco's approved open source policies, refer to
<http://wwwin.cisco.com/ops/ee/osrb/policies.shtml>

For questions about the policies, send email to the opensource-reviewboard@cisco.com.

The following are some of the important points to consider when using open source code in IOS software:

- 1** Any existing copyright information should not be deleted from or modified in the open source code.
- 2** Cisco copyright information should be added only to the files modified by Cisco engineers according to the existing copyright rules.

- 3 Before any open source files are added into Cisco IOS code, registration and approval from the IP Central team (<http://ipcentral.cisco.com>) are mandatory. The IP Central team will consult the development engineer and manager along with the attorney for the business unit if necessary before approving.
- 4 Approval of open source code depends on the license of the open source software, the platform on which the code runs, and how Cisco code interacts with the open source code.

For questions about the use of open source software, send email to opensource-specialists@cisco.com.

Cisco IOS Software Organization

This appendix contains information on the following topics:

- 12.2T Header File Location Changes for CDE
- Description of the Cisco IOS Subsystems
- Description of the IP Subsystems
- Description of the Cisco IOS Kernel Subsystems
- Alternatives to the Default CPU-Specific Object Directories

Note Cisco IOS code organization questions can be directed to their respective areas, such as interest-parser@cisco.com or interest-os@cisco.com, and to software-d@cisco.com because code organization crosses a lot of boundaries.

B.1 12.2T Header File Location Changes for CDE

The following header files have been moved in 12.2T to `../cisco.comp/.` for the future implementation of CDE (Component Development Environment), which is designed to support the seamless use of the same version of a component across multiple branches. See EDCS-182127 for more information on CDE and the `COMP_INC` macro.

Note The lists in this section may not be updated as soon as the CDE project is implemented.

B.1.1 Files Moved to `../cisco.comp/kernel/`.

Note For the following files, please check the `../cisco.comp/track` file for the full path name.

Note Be sure to replace `#include "xxx.h"` with `#include COMP_INC(kernel, xxx.h)` for these header files in 12.2T.

- 1 `clock.h`
- 2 `clock_format_string_lengths.h`

```
3 dqueue.h
4 ios_abort.h
5 ios_constants.h
6 ios_interrupts.h
7 ios_interrupts_platform.h
8 ios_io.h
9 ios_kernel_debug.h
10 ios_kernel_types.h
11 ios_kernel_util.h
12 ios_macros.h
13 ios_mem.h
14 list.h
15 logger_msg_def.h
16 mgd_timers.h
17 msg_sys.h
18 passive_timers.h
19 platform_clock.h
20 queue.h
21 queue_inline.h
22 target_ios.h
23 target_os.h
24 time_utils.h
25 timer_generic.h
26 unix/xxx.h
```

B.1.2 File Moved to .../cisco.comp/cisco/.

Note Be sure to replace #include "xxx.h" with #include COMP_INC(cisco, xxx.h) for these header files in 12.2T.

```
1 ciscolib.h
```

B.1.3 Files Moved to .../cisco.comp/ansi/.

Note Be sure to replace #include "xxx.h" with #include COMP_INC(ansi, xxx.h) for these header files in 12.2T.

```
1 ctype.h
2 dirent.h
```

```
3 errno.h  
1 fcntl.h  
2 gcc-2.95/xxx.h  
3 limits.h  
1 reent.h  
2 setjmp.h  
3 stdarg.h  
4 stdio.h  
5 stdlib.h  
6 string.h  
7 sys/xxx.h  
8 time.h  
9 unistd.h  
10 va-sparc.h
```

B.1.4 Files Moved to Newly Created .../cisco.comp/rc-tools/.

Note Be sure to replace `#include "xxx.h"` with `#include COMP_INC(rc-tools, xxx.h)` for these header files in 12.2T.

```
1 enum_def.h  
2 enum_template.h  
3 example_enum.rc  
4 self_include.h
```

B.1.5 Files Moved to .../cisco.comp/target-cpu/.

Note Be sure to replace `#include "xxx.h"` with `#include COMP_INC(target-cpu, xxx.h)` for these header files in 12.2T.

```
1 68k/xxx.h  
2 cisco_cpu.h  
3 cpu.h  
4 cpu_types.h  
5 intel/xxx.h  
6 isrhog_api.h  
7 mips/xxx.h  
8 posix_types.h  
9 ppc/xxx.h and ppd/xxx/xxx.h
```

10 sparc/xxx.h
11 target_cpu.h

B.2 Description of the Cisco IOS Subsystems

The rest of 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>.

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	AppleTalk, AppleTalk fast switching, and AURP
shr_atalkmib.o	
shr_atalktest.o	
shr_atfast_les.o	
ataurp.o (Release 11.2 only)	
atcp.o (Release 11.2 only)	
shr_smrp.o	AppleTalk Simple Multicast Routing Protocol (SMRP)
shr_smrptest.o	
shr_atsmrp.o	
at_smrpfast.o	
at_smrpfast_les.o	
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
vinesfast_les.o	Banyan VINES protocol support
shr_vines.o	
shr_vinesmib.o	
shr_vinestest.o	
shr_tarp.o	Target ID for the Address Resolution Protocol (ARP)
shr_bgpmib.o	Border Gateway Protocol (BGP) and MIB
shr_bgp.o	

Table B-1 Cisco IOS LAN Protocol Subsystems (continued)

Subsystem	Description
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
dncfast_les.o	DECnet
shr_decnet.o	
shr_decnetmib.o	
shr_dncnv.o	DECnet Phase 4-to-Phase 5 conversion
dhcp_client.o (shr_dhcp.o in Release 11.1)	Dynamic Host Configuration Protocol
shr_egp.o	Exterior Gateway Protocol (EGP)
shr_egpmib2.o	
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	IP fast switching
ipfast.o	
ipfast_les.o	
shr_iprouting.o	Generic IP routing functions
shr_ipservices.o	Basic IP services, including the TCP driver and
shr_tcpmib2.o	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

Table B-1 Cisco IOS LAN Protocol Subsystems (continued)

Subsystem	Description
shr_hpprobe.o (Release 11.2 only)	IP host functions
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_iparp.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)	
udp_flood_fs.o	UDP fast-switch flooding
ipnacl.o (Release 11.2 only)	IP named address lists
shr_ipmib2.o	IP multicast
shr_ipmroutemib.o	
shr_igmpmib.o	
shr_ipmulticast.o	
ipmfast_les.o	
rsvp.o	Resource Reservation Protocol and MIB
rsvpmib.o	
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	Novell IPX
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)	
shr_ipxcompression.o	IPX compression
shr_ipxnhrp.o	IPX Next Hop Routing Protocol (NHRP)
shr_ipxnlp.o	IPX NetWare Link Services Protocol (NLSP) and MIB
shr_isis.o	Intermediate System-to-Intermediate System (IS-IS) dynamic routing protocol
shr_isis_clns.o	
shr_isis_ip.o	
shr_isis_nlsp_debug.o	
shr_eon.o	EON-compatible ISO CLNS-over-IP tunneling

Table B-1 Cisco IOS LAN Protocol Subsystems (continued)

Subsystem	Description
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
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

Table B-2 Cisco IOS WAN Protocol Subsystems (continued)

Subsystem	Description
shr_smds.o	SMDS
shr_snapshot.o shr_snapshotmib.o	Snapshot routing
shr_v120.o	V.120 protocol over ISDN
shr_lapbmib.o	X.25
shr_pad.o	
shr_x25.o	
shr_x25mib.o	

Table B-3 Cisco IOS Bridging Subsystems

Subsystem	Description
shr_rsrpbmib.o shr_bridge_rsrp_bui.o	Remote source-route bridging (RSRB)
fastsrp_les.o	Source-route bridging (SRB)
shr_srbmib.o srpbmib.o (Release 11.2 only)	
shr_bridge_sr.o	
shr_bridge_srb_bui.o	
srpbcore.o (Release 11.2 only)	
tshr_bridge_t.o shr_bridge_t_bui.o	Transparent bridging and MIB
tbridge.o	
tbridge_les.o	
shr_tbmib.o	
bridge_t_cmf.o (Release 11.2 only)	
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 key
shr_lat.o	Local-area transport (LAT)

Table B-4 Cisco IOS Communications Server Subsystems (continued)

Subsystem	Description
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	Protocol translation
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	
shr_radius.o	Livingston's "Radius" network authentication protocol
shr_tacacs_plus.o	TACACS+
shr_tn3270.o tn3270s.o (Release 11.2 only)	TN3270 and MIB
shr_tsmib.o (Release 11.2 only)	
shr_xremote.o	XRemote

Table B-5 Cisco IOS Utilities Subsystems

Subsystem	Description
shr_ifmib.o	Interfaces MIB
shr_des.o	Data Encryption Standard (DES)

Table B-6 Cisco IOS Driver Subsystems

Subsystem	Description
sub_pcibus.lnm.o	Network Management for AccessPro ISA
sub_pcibus.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 2512 access servers
sub_hub.o	Driver for Cisco 2500 series hubs
sub_brut.o	Console/auxiliary port driver for Cisco 2500 series

Table B-6 Cisco IOS Driver Subsystems (continued)

Subsystem	Description
shr_ether.o	Generic Ethernet driver
shr_ethermib.o	
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	Generic Token Ring driver
shr_trmib.o	
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	Cisco Discovery Protocol (CDP)
shr_cdpmib.o	
cdp_ncp.o (Release 11.2 only)	
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	Remote monitoring
shr_rmonlite.o	
sr_rs232mib.o	SNMPv2 bilingual agent code
sr_mempoolmib.o	
shr_syslog_history.o	Syslog messages
shr_syslogmib.o	

Table B-8 Cisco IOS VLAN Subsystems

Subsystem	Description
vlan_les.o	Virtual LAN
ieee_vlan.o	
isl_vlan.o	
shr_vlan.o	

Table B-9 Cisco IOS Kernel Subsystems

Subsystem	Description
shr_core.o (Release 11.2 only)	Cisco IOS kernel. For Release 11.2, the kernel is divided in platform-independent and platform-dependent portions. <code>shr_ukernel.o</code> contains the essentials necessary for the scheduler to operate, including memory management, list support, timer support, subsystem and registry support, and basic clock support. <code>shr_core.o</code> contains other things that were in <code>sub_kernel.o</code> , 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.
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)	

Table B-10 Cisco IOS IBM Subsystems

Subsystem	Description
shr_dlur.o	Advanced Peer-to-Peer Networking (APPN)
shr_appn.o	
shr_appnmib.o	
shr_appnutil.o	
sub_bsc.o	Bisync
shr bstun.o (Release 11.2 only)	Bisync serial tunnel
shr bstunmib.o (Release 11.2 only)	
sub bstun.o (Release 11.1 only)	
sub bstunmib.o (Release 11.1 only)	
shr_dlcsw.o	CLSI
shr_dlc_base.o	
shr_vdlc.o (Release 11.2 only)	
fastdlsw_les.o	Data Link Switching (DLSw) and MIB
shr_dlsw.o	
sr_cdlswmib.o	
shr_dspu_ui.o	Downstream PU (DSPU)
shr_dspumib.o	
shr_ibmmm.o	IBM Network Management protocol

Table B-10 Cisco IOS IBM Subsystems (continued)

Subsystem	Description
shr_lack.o	Local Acknowledgment
shr_lanmgr.o	Token Ring LAN manager
shr_lanmg_ui.o	
shr_llc2.o	LLC2
ncia.o (Release 11.2 only)	Native Client Interface Architecture (NCIA)
ncia_ui.o (Release 11.2 only)	
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
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

Table B-11 Cisco IOS Library Utility Subsystems (continued)

Subsystem	Description
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 CLNS and 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 number generator routines
random_fill.o	
range.o	Routines for range arithmetic
regexp.o	Routines for regular expressions
regexp_access.o	
regsub.o	
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
_toupper.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

Table B-12 Cisco IOS ANSI Library Subsystems (Release 11.2 only) (continued)

Subsystem	Description
iscntrl.o	Determine whether argument is a control character
isdigit.o	Determine whether argument is an ASCII decimal digit
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

Table B-12 Cisco IOS ANSI Library Subsystems (Release 11.2 only) (continued)

Subsystem	Description
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
badbdc.o	Return nonzero if BCD string has a bad entry
bcdtochar.o	Convert BCD string to a character
bcmpl_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

Table B-13 Cisco IOS Cisco Library Subsystems (Release 11.2 only) (continued)

Subsystem	Description
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.3 Description of the IP Subsystems

This section details the object files in the following IP routing protocol subsystems for Cisco IOS Release 11.1.

- IP Host Subsystem
- IP Routing Subsystem
- IP Services Subsystem

B.3.1 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	Basic nonrouting core IP services
ip_init.o	
ipsupport.o	
ip_debug.o	
ip_setup.o	
ip_test.o	
msg_ip.o	
ipname.o	
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 command-line parsing
ip_actions.o	
ip_chain.o	
ipoptions.o	IP security options
msg_ipsecure.o	
ipoptparse.o	
icmp.o	Internet Control Message Protocol (ICMP)
icmpping.o	
path.o	IP routing using ICMP redirects
iptrace.o	Traceroute
ip_arp.o	ARP and Reverse ARP
rarp.o	
probe.o	HP Probe (Hewlett-Packard's version of ARP)
probe_chain.o	
bootp.o	BOOTP boot code
gdpcclient.o	Gateway Discovery Protocol (GDP)
gdpcclient_chain.o	
gdp.o	
gdp_chain.o	
tftp.o	Trivial File Transfer Protocol (TFTP)
tftp_server.o	
tftp_chain.o	
tftp_debug.o	
tftp_util.o	
tacacs.o	TACACS
xtacacs.o	
tacacs_chain.o	
msg_tacacs.o	
ipaccess1.o	IP accounting
ipaccount.o	
ipaccount_chain.o	

Table B-14 IP Host Subsystem Object Files (continued)

Object File	Description
tcpfast.o	TCP core services
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	
tcpcompress.o	TCP compression
tcpcompress_chain.o	
rcp.o	Remote copy (rcp) and remote shell (rsh)
rsh.o	
msg_rcmd.o	
telnet.o	Telnet
telnet_debug.o	
msg_telnet.o	
ipaccess2.o	IP access lists
ipaccess_chain.o	
ipalias.o	IP aliases
ipalias_chain.o	
ip_snmp.o	SNMP support
msg_snmp.o	

B.3.2 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	Common IP routing routines
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	
ipigrp2.o	IP IGRP
radix.o	Radix trees
ipradix.o	
msg_radix.o	
irdp.o	ICMP router discovery protocol
irdp_chain.o	
community.o	IP community list-related functions
standby.o	Host Standby Routing Protocol (HSRP)
msg_standby.o	
ipmobile.o	IP host mobility protocol
ipmobile_chain.o	
nhrp.o	Next Hop Routing Protocol (NHRP)
nhrp_cache.o	
nhrp_vc.o	
msg_nhrp.o	
ipnhrp.o	

B.3.3 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
dnsix_nat.o	
dnsix_chain.o	
dnsix_debug.o	

B.4 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.4.1 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	Scheduler
sched_compatibility.o	
sched_test.o	

B.4.2 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.4.3 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	Support for generic static maps
static_mapchain.o	

B.4.4 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	EXEC command support
command2.o	
command_chain.o	
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
latgroup.o	LAT group code handling
setup.o	
exec.o	EXEC functions
exec_chain.o	
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.4.5 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	System authentication, authorization, and accounting functions
aaa_acct.o	
aaa_chain.o	
keyman.o	
connect.o	Connection management services
connect_chain.o	
hostname.o	Host command support
async.o	Asynchronous port support
async_chain.o	
async_debug.o	
login.o	Old system authentication (replaced by aaa object files)
login_chain.o	
modemsupport.o	Modem support
monitor1.o	ROM monitor support
ttycon.o	Terminal services
ttysrv.o	
ttystatem.o	

B.4.6 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	Common access list support
access_chain.o	
if_groups.o	Interface groups
if_vidb.o	Virtual IDB support
interface.o	Functions for manipulating software and hardware IDB
interface_api.o	structures
linkdown_event.o	Ethernet, Token Ring, and HDLC link-down handling for SNA 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	Generic network support, including keepalives, hold queue
network_debug.o	management, interface commands and IDB commands
pak_api.o	Packet interface API
priority.o	Priority queueing
priority_chain.o	

Table B-22 Core Router Subsystem Object Files (continued)

Object File	Description
trace.o	Support for trace command
compress_lzw.o	Compression support
config_compress.o	

B.4.7 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	Buffer management support
chunk.o	
element.o	Memory management support
free.o	
logger.o	System logging support
logger_chain.o	
print.o	System print services
printf.o	
region.o	Region management services

B.4.8 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	Basic system clock and tim support routines
clock_guts.o	
clock_util.o	
mgd_timers.o	Timer support
time_utils.o	
timer.o	
timer_chain.o	

B.4.9 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 and subsystem support
registry.o	
registry_debug.o	
subsys.o	

B.4.10 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	Network configuration and loading support
coverage_analyze.o	
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 as 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
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	Per-process stack creation, manipulation, and monitoring routines
stacks_68.o	

Table B-26 Miscellaneous Core Subsystem Object Files (continued)

Object File	Description
sum.o	Checksum support
util.o	Miscellaneous utility routines, including time-related services and case conversion

B.5 Alternatives to the Default CPU-Specific Object Directories

By default, object modules which are consistent across all platforms are kept in the default CPU-specific object directories. These directories have the format: obj-ccc (where ccc is the cpu type). Examples are obj-4k, obj-m860 and obj-68.

There are currently two alternatives to these default cpu-specific directories:

- **obj-ccc_64** This common object directory is used by those platforms which need 64 bit counters (and whose .o-s can't be shared with platforms which need 32-bit counters, since the IDB structure layout is different). This gives 64-bit counter support without affecting the low-end platforms. (New in 12.0S)

Note The 64-bit counter support isn't limited to the MIPS platforms (the 4K family). The next generation GSR RP (marketing calls it the PRP) uses a PowerPC processor and the obj-g4_64 directory for storing common object files. Currently GSR is present in 12.0S only, but they are going to be in 12.2S RLS6.

- **obj-ccc_OPT** This common object directory is used on platforms which have compiler switches enabled to reduce the image size (while sacrificing performance). See CSCdw26619 or ENG-167610. The tradeoff is required on platforms which were running out of flash space. For example, the 2600 series uses obj-m860_OPT and the 3600 uses obj-4k_OPT.

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. For more information on performance analysis, please see:

<http://wwwin-ses.cisco.com/performance/index.html>

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 (250 times 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.

Note Cisco IOS CPU profiling questions can be directed to the profilers@cisco.comalias.

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 Ticks

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 680x0 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.2.5.1 Notes for Dealing with CPUHOGs

Many of the CPUHOG messages are obscured by adding `process_may_suspend()` or `process_wait_for_event()` rather than fixing the underlying unscalability caused by poorly designed algorithms or data structures. While it's true that adding `process_may_suspend()`

or `process_wait_for_event()` has benefits, it's frequently not solving the underlying problem of an algorithm running longer than it needs to. This may result in severe problems later. In most cases, CPUHOGs are caused by algorithms that may have been fine when written but that don't scale to meet today's requirements. This is just a warning against blindly adding `process_may_suspend()` or other OS calls that potentially suspend process execution (see the *Cisco IOS API Reference*) to fix CPUHOG messages.

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 1** Configure the profiler to capture the data that you want. Specify a mode appropriate to the problem you are trying to solve.
- Step 2** 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 3** Pass the captured file, along with a symbol table matching the image running in the system, to the postprocessing program.
- Step 4** Scratch your head and mutter.
- Step 5** Repeat Step 1 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 680x0-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 Block

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 Task and 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 `flo_t_pi6`, you can find it in `sys/scripts/profile_cpu.[ch]`. This 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.

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 Release 11.3 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 Cisco IOS 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 “Basic IOS Kernel Services” 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 “Basic IOS Kernel Services” chapter in this manual.

Note Cisco IOS scheduler questions can be directed to the interest-os@cisco.com alias.

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

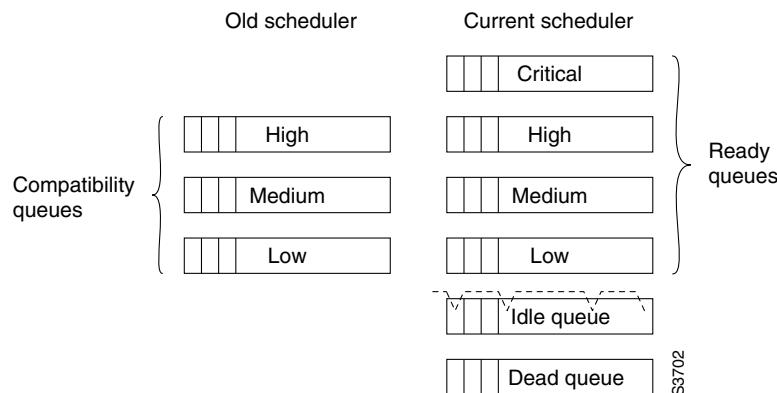
D.2.1 Scheduler Queues

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

Figure D-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.

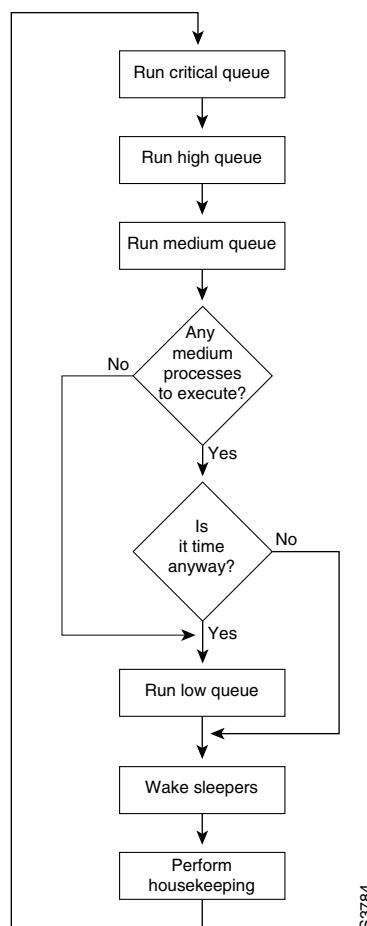
D.2.2.1 Overall Scheduler Queue Operation

The overall scheduler queue operation is as follows. Figure D-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.

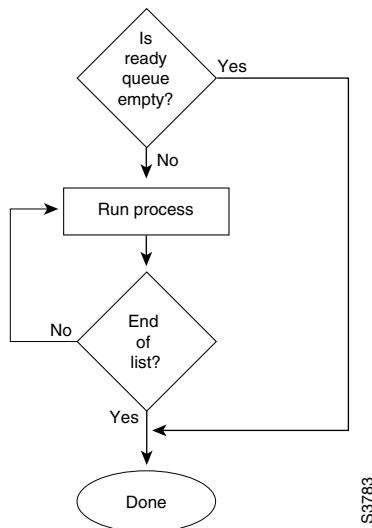
Figure D-2 Overall Scheduler Queue Operation



S3784

D.2.2.2 Critical-Priority Scheduler Queue Operation

Processes on critical-priority queue are handled by the scheduler immediately, as illustrated in Figure D-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. Figure D-4 illustrates this operation.

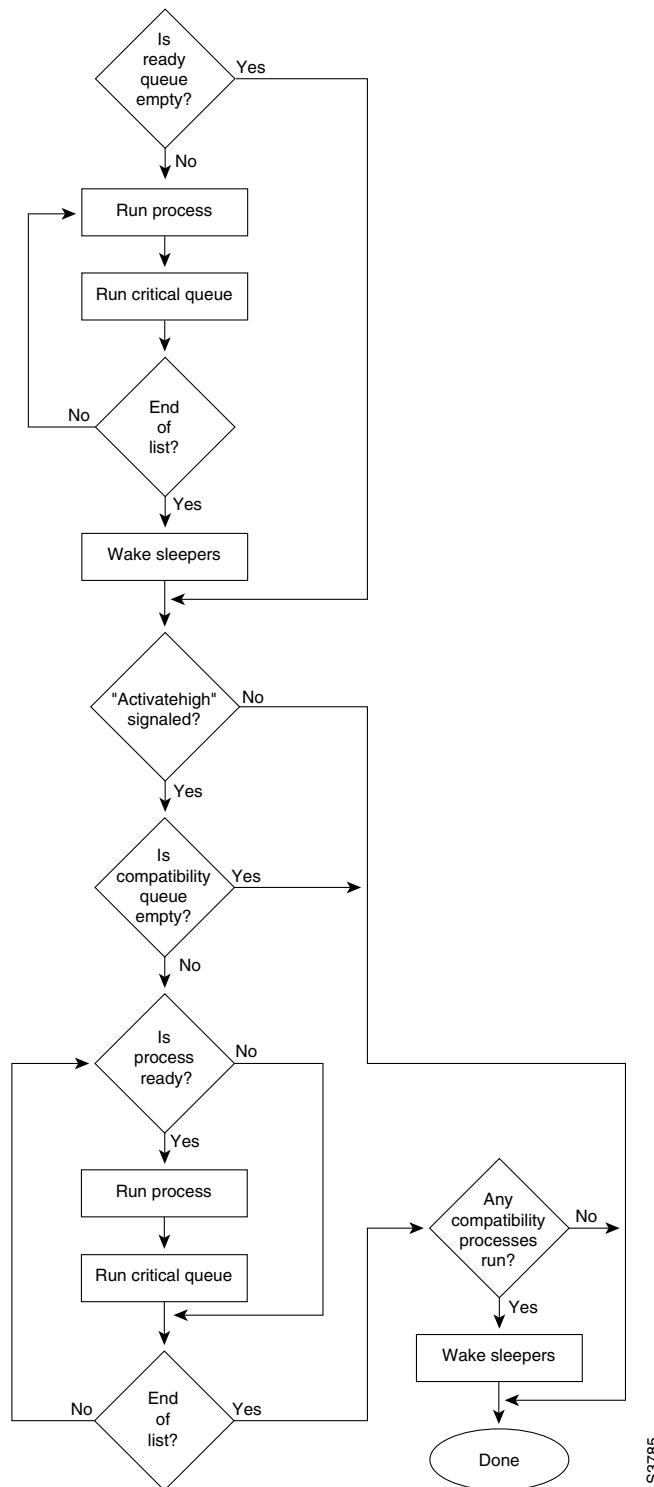
Check the Ready Queues

- 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



S3785

D.2.2.4 Medium- and Low-Priority Scheduler Queue Operation

The operation of the medium-priority queue is as follows. Figure D-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.

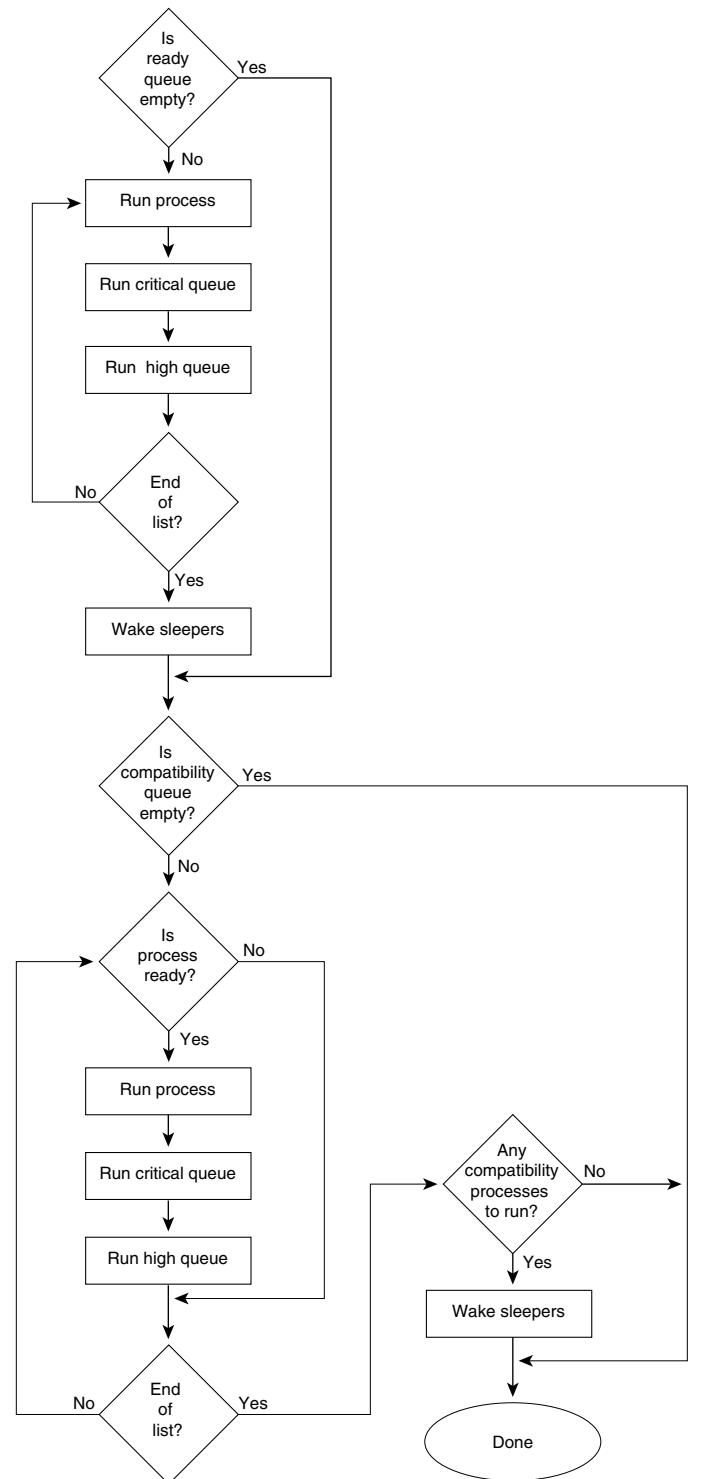
Check the Compatibility Queues

- 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



S3782

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 Release 11.3, support for these functions has been removed.

lists 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.1 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

<i>paddr</i>	Starting address of the process to execute.
<i>pp</i>	Parameter to the process.
<i>stack</i>	Size of the process stack in <i>words</i> . A value of 0 uses the default stack size.
<i>name</i>	Textual name of the process as it should appear in all output.
<i>ttynum</i>	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

<i>pid</i>	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.2 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 if 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.3 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.4 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

<i>priority</i>	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_count_free\(\)](#)
[process_get_priority\(\)](#)
[process_set_arg_num\(\)](#)
[process_set_arg_ptr\(\)](#)

D.5 `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_count_free()`

D.6 `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_count_free\(\)`](#)

Branch Integration & Sync Processes

E.1 Overview

This appendix summarizes the life cycle of a Cisco IOS branch, beginning from its conception (branch pull) to the end of its life (collapse). It covers the various phases that are typical in the life cycle of a Cisco IOS branch such as incremental syncs, reparents, terminal syncs, and collapse.

This appendix includes information on the following topics:

- Terms in Branch Integration
- Branch Life Cycle
- Branch Modeling
- Reparenting
- Porting
- Cloning
- Collapsing
- Syncing
- Roles and Responsibilities of the Stake Holders during a Sync

Note Cisco IOS branch integration and sync process questions can be directed to the flo-sync@cisco.com (12.2, 12.2T, 12.2 PI sync information alias), geo-sync@cisco.com (12.3, 12.3T, and 12.3 PI sync information alias), and sync-tech-team@cisco.com (group focused on improving the Cisco IOS sync process) aliases.

E.1.1 Intended Audience

This appendix is intended as a quick summary for both the release groups (PMs, “syncmeisters”, and the build team) and the development team working on a Cisco IOS branch. A basic knowledge of the ClearCase (cc) tools in use at Cisco is assumed. If not familiar with any terms or concepts in this appendix, consult

<http://wwwin-swtools.cisco.com/swtools/clearcase/>

E.2 Terms in Branch Integration

There are many common terms frequently used by the sync group. These terms are keywords; they are specific to the context of using the CC Tools at Cisco. This section summarize the definitions of commonly used terms in branch integration, including sync processes.

Note This document uses the standard way of depicting branches: the child branches are drawn below the parent branches.

Types of Branch

baseline

A release branch (or a stable branch) used as the starting point for new feature development.

preintegration branch

A preintegration branch is a branch used to integrate a few features before collapsing it to a release branch. The reasons we do it are that we get to test the features in a “safe” environment before code is put on a release branch, and that the release branches are kept relatively clean as only tested code gets committed.

integration branch

The ClearCase branch to which changes are committed. This type of branch is used for shared development by a team working closely on a project or a release. Integration branches are permanently stored in ClearCase.

task branch

A ClearCase branch that is owned and controlled by an individual user. This type of branch is short-lived and is removed once the changes have been committed into an integration branch.

Branch Integration Operations

clone

Cloning is the process that is used to exactly replicate a branch. The cloned branch will have the same parent (REF_LABEL) and contents as the original branch. This is only done in some special cases.

collapse

Collapsing is the process of delivering all the changes on a child branch (feature additions and fixes) back to its parent. (Right before the collapse, a terminal sync is performed and the latest code on the parent is integrated into the child. See “Types of Sync” for a definition of “terminal sync.”)

merge

This is the generic term for the act of combining different strands of development, with the specific meaning being determined by the context. In Cisco IOS development, a combination of Cleartool **findmerge** and RCS **merge** is used to merge code. These tools work on the affected elements one at a time to complete the merge. Merging plays a key role in all branch integration processes.

patch

This is a process in which the file and directory diffs that are present in a “patch file” are applied to a branch. The patch file is typically produced by the **cc_diff** tool and the patch is done by the **cc_patch tool**.

port

A port incorporates changes made in a source branch or a label into a target branch. Depending on the situation, either all changes or a subset of that is incorporated into the target. In all ports, the entire set of source and target branches are unrelated; that is, one is not a direct descendant of the other. The changes from the source or label are either merged into the target (using Cleartool **findmerge** and RCS **merge**) or patched (using **cc_diff** and **cc_patch**).

reparent

Reparenting is the process that changes the parent, and hence the ancestor history, of a branch. A branch is reparented in two steps. First the parent attribute or the child is changed to reflect the new parent. Then a reparent sync is done to a label on the new parent. The label on the new parent should include all the changes made in the old parent. The main requirement for a legal reparent is that the *new parent* should contain all the changes made in the *old parent*.

Most often, a branch is reparented to its grandparent (after the collapse of the current parent), or to a sibling (to merge in the sibling’s changes without waiting for a collapse).

Consider a CC view of a child branch containing a number of elements (files or directories). Only those elements that have been changed on the parent will be affected by the sync. For elements that branched onto the child, this is achieved by merging the child’s version with the parent contributor’s version. For elements not branched onto the child, known as “bleed-through elements”, this is achieved by moving the child’s REF_LABEL, which then affects the config spec for new or updated views of the child branch.

Conceptually, this is very similar to running a “**prepare -m LABEL**” on a task branch.

Types of Sync

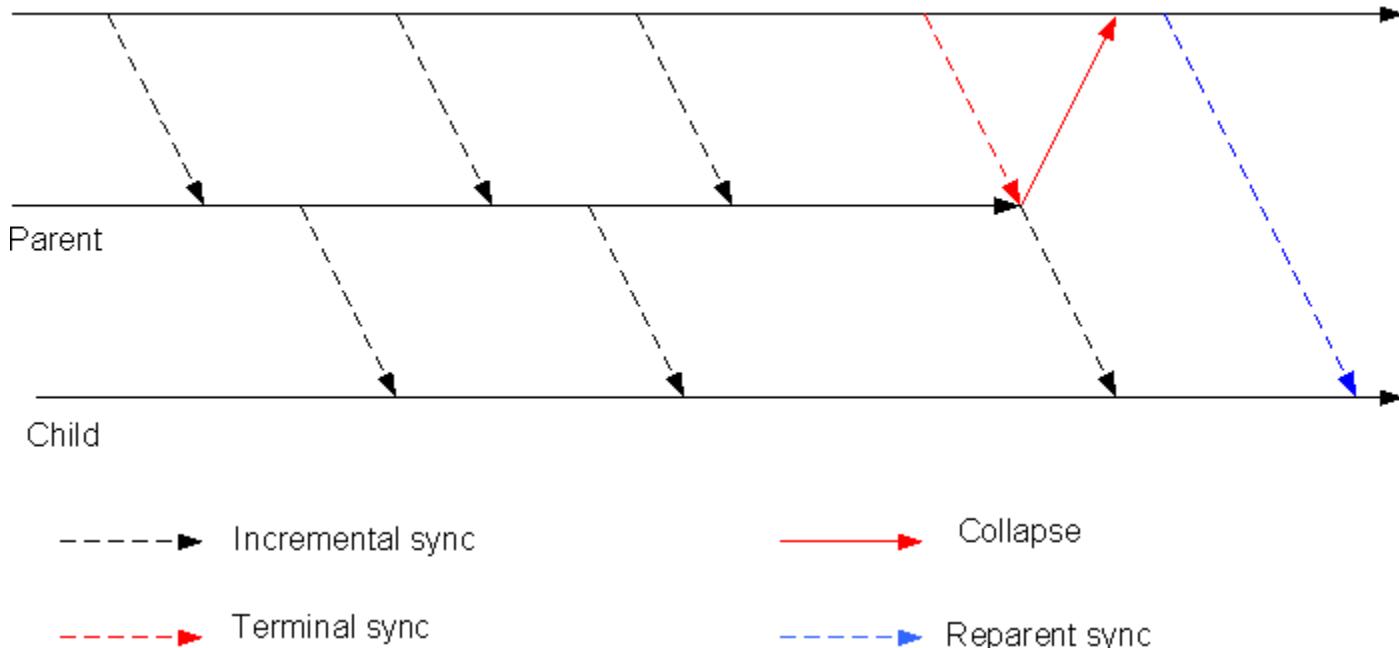
The four different types of sync are: incremental, reparent, and terminal.

incremental sync

A sync performed as a stand-alone process to update the child with the parent’s changes. The changes can include bug fixes and/or new features.

Figure E-1 Incremental Sync

Grandparent



-----► Incremental sync

-----► Collapse

-----► Terminal sync

-----► Reparent sync

reparent sync

A sync performed as part of the reparent process after the parent attribute of the branch has been changed. In this type of sync, the child branch, previously sync'd to a label on the old parent, is sync'd to a label on the new parent. The reparent sync completes the reparent process and ensures that the child is able to get code from the new parent.

sync

Sync is the process of incorporating the changes that have been made in the parent branch as of a particular label into a child integration branch. Afterwards, the child is said to be “sync’d” to the parent as of that label, and that label is the child’s new REF_LABEL (attribute). For example, in Figure E-1, one would say, “flo_t is sync’d to florida as of florida label LF3 and LF3 is now flo_t’s REF_LABEL”. The developers on the child branch flo_t will now see all the new functionality and bug fixes made to the parent florida as of the new REF_LABEL, LF3, rather than only as of the old REF_LABELs, LF1 or LF2.

terminal sync

A sync performed as part of the collapse process. This sync is typically done to the latest available label on the parent. This is the last sync performed on a branch before it is collapsed into the parent.

Elements

base contributor

This is the version of the element that is used as the “common point” for the merge. Typically, this is the version seen in the old REF_LABEL.

bleed-through

A bleed-through file is an element or file that is not branched on the child, that is, has not been modified on the child. For these elements we see the parent's or ancestor's version because bleed-through files are not merged during a sync.

child contributor

This is the version of the file on the child that contributes to create a merged file. **findmerge** checks out this version of the file to place the merged file.

config spec

The file used to specify the configuration used by a particular view.

configuration

A specific set of versions of source files. Normally, a configuration is somewhat permanent in that there is a desire to be able to recreate the configuration at any time in the future. A good example of the use of configurations is the weekly builds of the main release trains performed by the build group.

contributors

These are the three versions of an element that together create a merged file. During a sync, all changes made between the base contributor and parent contributor are incorporated into the child contributor to create the merged file. Figure E-2 shows the three contributors. The parent contributor is foo.c@@/main/7, the base contributor is foo.c@@/main/5, and the child contributor is foo.c@@/main/branch/3. (The base contributor can be a blank file, but then, that is the contributor.)

element

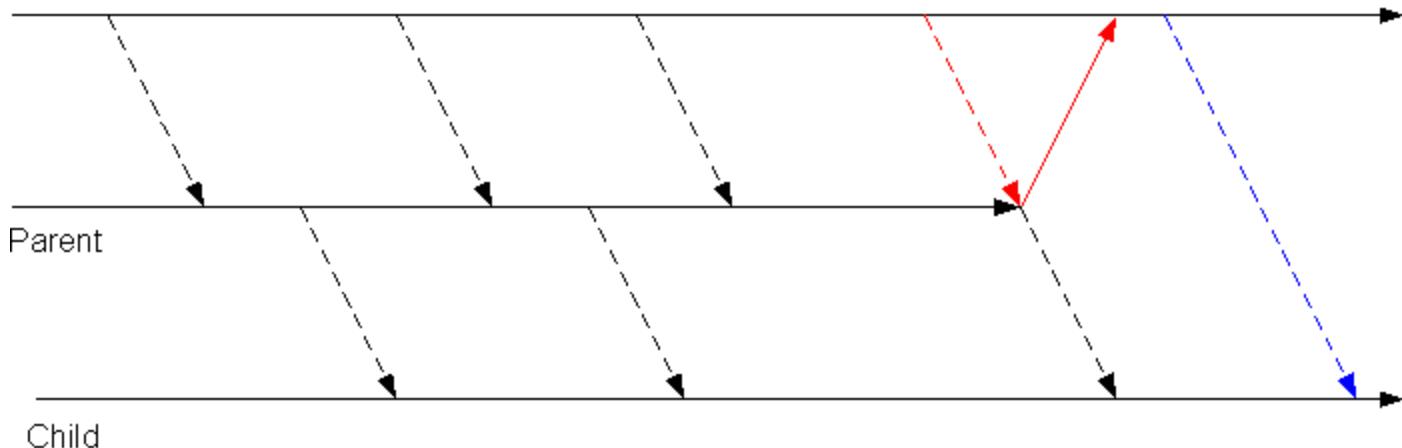
An element is an object in ClearCase which encompasses a set of versions organized into a version tree. In ClearCase, an element includes both files and directories.

parent contributor

This is the version of the element on the parent branch that contributes to the merge. This is the version of the element seen in the new **REF_LABEL** (attribute on the child branch).

Figure E-2 The Three Contributors

Grandparent



-----► Incremental sync

→ Collapse

-----► Terminal sync

-----► Reparent sync

Labels

base label

The base label is a floating label (i.e. the opposite of a fixed label) that is used to dynamically track the ref label of the child branch. As part of each sync, the base label is moved so that it corresponds to the new ref label. By convention, the base label is the same as the branch name, in ALL CAPS (e.g., the base label for projectx is PROJECTX).

label

A label is a mechanism provided by ClearCase to assist in capturing configurations. A label is added to the individual versions of a configuration.

reference label (REF_LABEL)

The label, (for example, the current REF_LABEL for geo_t is: v123_3_2U) on the parent branch to which the child branch is currently sync'd. A REF_LABEL attribute is attached to the child branch for tracking this point.

stable label

A label on a branch that indicates that the branch content up to that point has undergone appropriate testing and validation, as determined by the branch manager. These labels are usually the recommended sync points for child branches. The criteria for assigning a stable label depends on the person or team managing the branch.

E.3 Branch Life Cycle

Typically, an integration branch is pulled when the development team is ready to start the coding phase of the software lifecycle. Which parent branch and which label used for pulling a branch depends on these things: the targeted customer, the time frame, and any dependencies on other features. Also, the label from which the branch is pulled must be stable (tested and validated). Each branch has various stakeholders, (such as release PMs, sync engineers, build engineers) who maintain and control the branch destiny. Once the branch has been pulled, incremental syncs are performed on a regular basis to obtain bug fixes and new code from the parent. The frequency of the incremental syncs depends on the volume of code changes made on the parent since the last sync'd label.

Throughout its lifecycle, the branch may have the need to undergo some hierarchical changes, such as reparenting or porting. These operations depend on the nature of the parent and the requirement of the feature in the integration branch. When the feature is ready to be committed, the branch is then scheduled for the terminal sync and collapsed to its parent. Afterward, the branch will be marked “END OF LIFE” when the branch will no longer allow new code commits.

The following two diagrams show the common scenarios that a Cisco IOS branch experiences in its lifetime. The first one, Figure E-3, shows a typical branch life cycle and the second one, Figure E-4, shows a life cycle that includes porting.

Figure E-3 Typical Branch Life Cycle

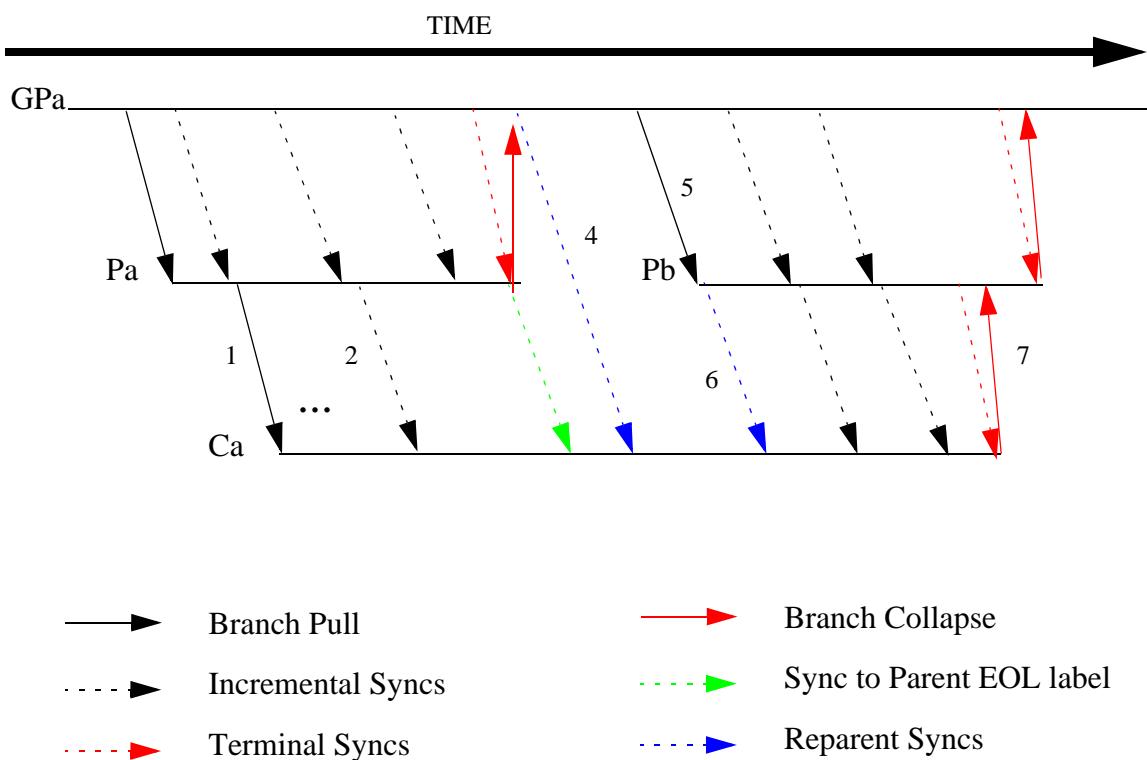


Figure E-3 shows a typical branch life cycle which includes the following steps which correspond to the numbers in the diagram:

- Step 1** First, the branch Ca is pulled from a stable label on the parent, Pa (number 1 in Figure E-3).
- Step 2** Then incremental syncs are being performed on Ca to bring down bug fixes and new code from the parent (number 2 in Figure E-3).
- Step 3** Then, the Pa branch is collapsed to its parent, GPa. GPa is the grand parent branch, that is, it is the parent of Pa.
- Step 4** This step happens after the collapse. The Ca branch needs to do a sync to the EOL label of Pa, before it is reparented to GPa. This is done by changing its parent attribute and then performing a reparent sync.
- Step 5** Later in the branch life, a new branch named Pb is pulled from GPa.
- Step 6** Then, Ca again needs to be reparented.
- Step 7** When the branch Ca is ready to be committed, a terminal sync and a collapse are performed.

Note that Figure E-3 is a simplified conceptual chart since there are branches that will be reparented numerous times before they are eventually collapsed.

Depending on the situation, the operations are done either by a Development Engineer or a full-time Sync Engineer. Section E.10 “Roles and Responsibilities of the Stake Holders during a Sync” has a detailed description of the roles and responsibilities.

Figure E-4 Branch Life Cycle With Porting

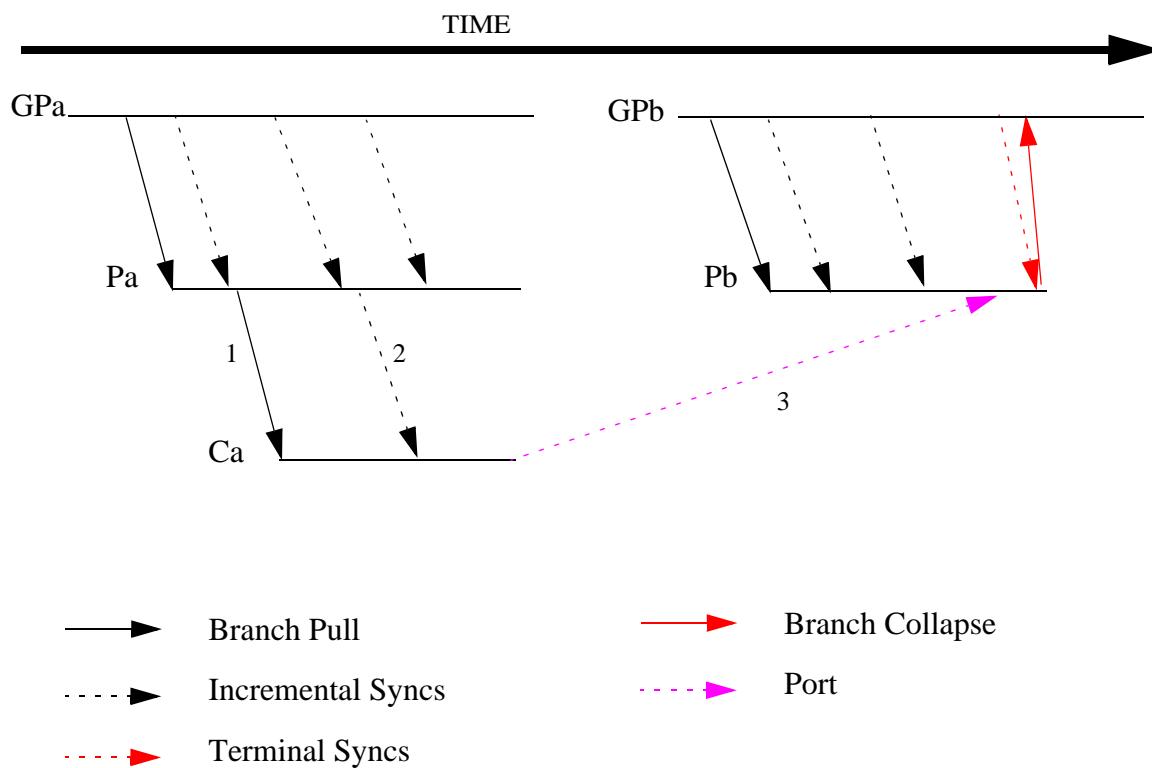


Figure E-4 shows that a branch with porting includes the following steps:

- Step 1** Branch Ca is initially pulled from the parent, Pa.
 - Step 2** Incremental syncs are performed regularly. Development on branch Ca has moved to a large extent. This means that a lot of development has occurred on the branch Ca. Engineers have made a lot of changes on the branch Ca.
 - Step 3** Later on, the release team or the development team decides that the features in Ca should be committed to another branch, Pb, which has no common code base with Ca's current parents, Pa, GPa. In this case, Ca needs to be ported to Pb, and then, regarding to Ca's life, it is collapsed to GPb.

E.4 Branch Modeling

Within Cisco, we use a logical branching model (sparse branching) in our VOB groups. What this means is that when we create an integration branch, we do not create a physical branch for each and every file in the relevant repository. Instead, we rely on cspec bleed-through to select an appropriate version on an ancestor branch. Elements (files or directories) are on demand, that is, whenever a file is modified on a child.

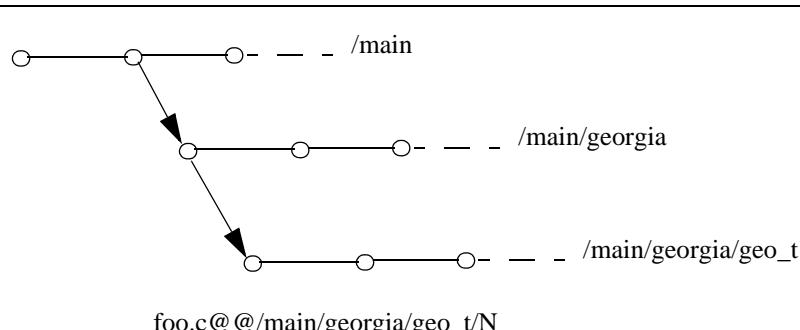
The advantages of this branching scheme are:

- fewer branched elements and a smaller database
 - fewer multi-site operations
 - faster Clearcase operations, like **find**, **merge**, etc., because we have fewer objects to search through
 - faster and safer syncs. Without sparse branching, you would have to merge **every** element that's been changed on the parent and this could be error-prone.

A side-effect of sparse branching is that the physical/actual branch structure of any given element may not be the same as the logical branch structure of the VOB group.

As a trivial example, consider branch geo_t which is logically parented on /main/georgia. If we look in a CC view of geo_t at the particular element named foo.c, the version selected by the view might have any of the forms shown in Figure E-5, Figure E-6, Figure E-7, or Figure E-8:

Figure E-5 File Changed by both geo_t and georgia



N means any version number - 1, 2, etc.

Figure E-6 File Changed by geo_t Branch, but Not by georgia Branch

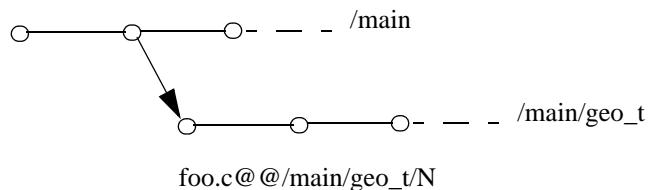


Figure E-7 File Changed by georgia Branch, but Not by geo_t Branch

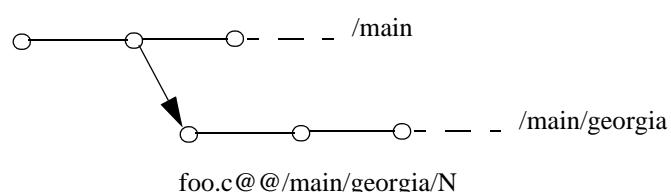
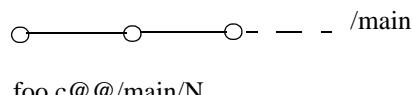


Figure E-8 File Changed by neither the georgia nor the geo_t Branches



The possible permutations depend on the depth of the logical branch structure, i.e., $2^{**(\text{depth}-1)}$. For a branch like geo_t_pi2, which has a depth of 4 in its logical branch structure, there are $2^{**3} == 8$ (two to the three equals eight) possible physical branch configurations that might show up in a view of geo_t_pi2.

There are no bad results from this side effect.

E.5 Reparenting

Reparenting is the process of changing the meta-data of the source branch, to merge in the content/code changes of the new parent, provided the new parent has all the dependencies required by the source branch. The process involves two steps:

- Step 1** Changing the source branch's meta-data
 - Step 2** Performing a sync to the new parent (reparent sync)

E.5.1 Why Reparent?

Reparenting is performed for various reasons. The main reason for doing a reparent is to make available the features and bug fixes in the new parent on the source branch.

In the case where the current parent branch has been collapsed to its parent, the branch should be reparented to its grandparent to continue picking up bug fixes and new code.

If the branch needs to be released from a different baseline, then the child could be reparented to the new baseline. Reparenting can be done from one parent to a descendant of that parent.

E.5.1.1 Reparenting Considerations/Requirements

When reparenting, the new parent needs to be a superset of the old parent, which means that the new parent's sync point must be the same or later than the old parent's sync point. Otherwise, some of the dependencies of the source branch may not exist on the new parent/ancestor.

E.5.1.2 Reparenting Advantages/Disadvantages

Advantages: The branch can easily pick up the code of the new parent without doing the actual porting. We accomplish this by simply changing the parent attribute and performing a reparent sync.

Disadvantages: Due to the change in the ancestor history, you could end up picking up a lot of changes if there is a huge divergence between the old baseline and the new baseline. Also, in some cases, Clearcase will have difficulties determining the common ancestors, and, hence, will create spurious merge conflicts.

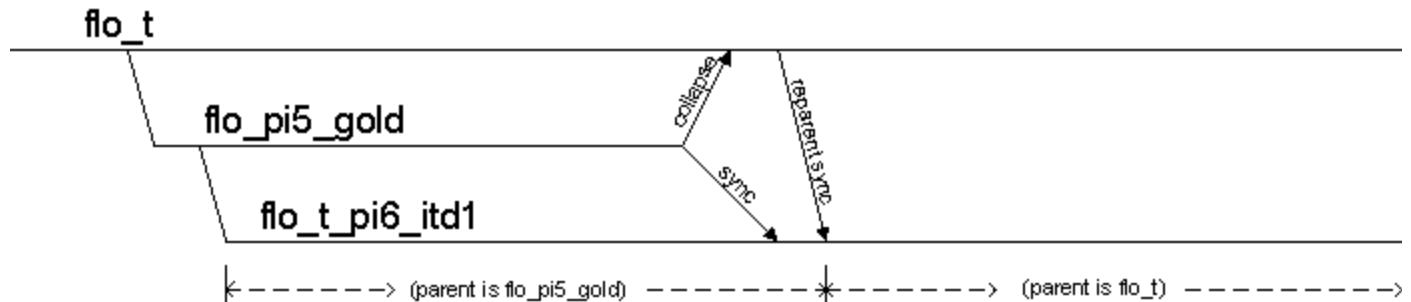
E.5.2 Reparenting Scenarios

The following scenarios show when a branch can or cannot be reparented.

E.5.2.1 Reparent to the Grandparent Branch

In Figure E-9, flo_t_pi6_itd1 should be reparented to flo_t after flo_pi5_gold collapses to flo_t.

Figure E-9 Example of Branch Reparented to Its Grandparent Branch



E.5.2.2 Reparent to a Sibling Branch

In Figure E-10, clynelish can be reparented to flo_11_e since flo_11_e is a superset of flo_isp.

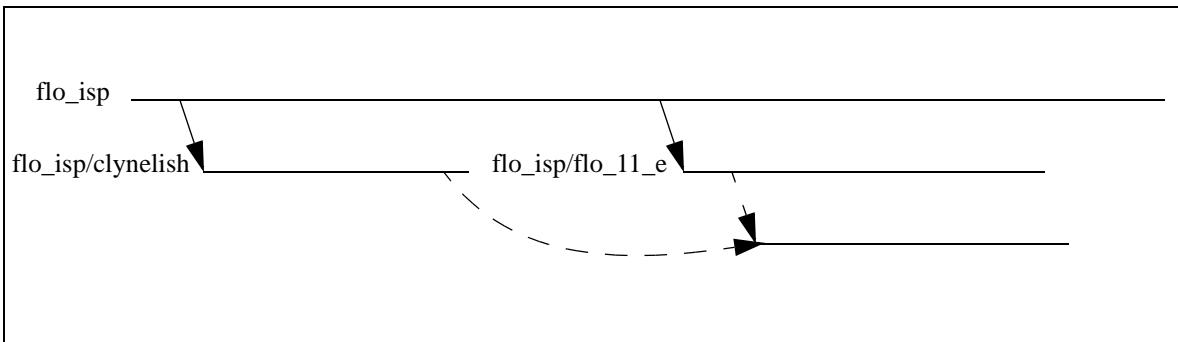
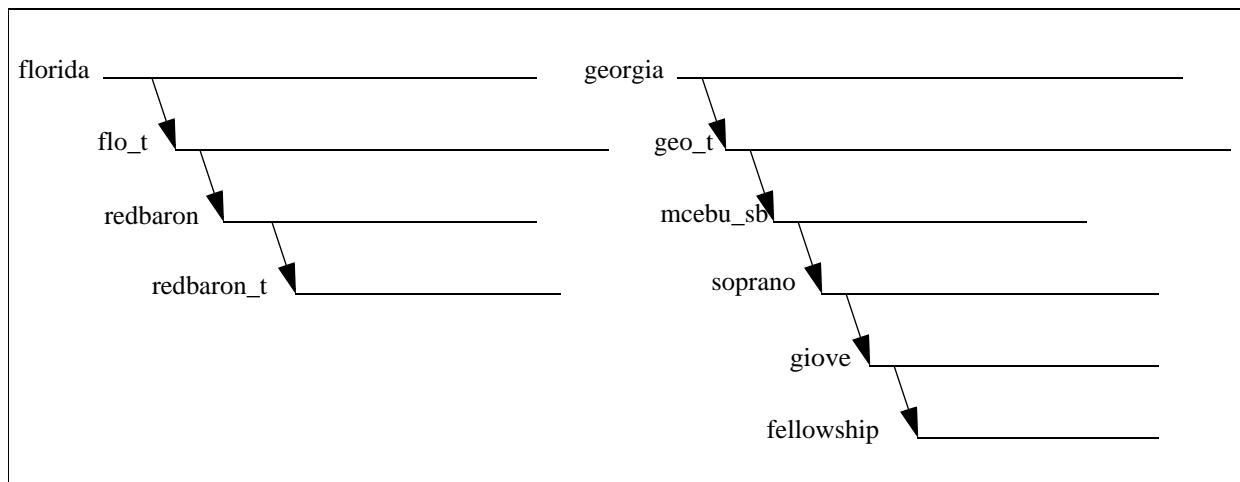
Figure E-10 Example of a Branch Reparented to Its Sibling Branch**E.5.2.3 Example of a Branch That Cannot Be Reparented**

Figure E-11 is an example of why a branch cannot be reparented to a certain other branch. Here `redbaron_t` cannot be reparented to `fellowship` since `redbaron_t` has some dependencies on `redbaron`. `redbaron` doesn't exist on any of the branch hierarchy under `georgia`. Hence, it is not a superset of `redbaron_t`'s code base and therefore `redbaron_t` cannot be reparented to it.

Figure E-11 Example of a Branch Not Reparented**E.6 Porting**

Porting is the process of transporting changes (features or fixes) that have been made on the source branch between two labels, to a new or existing branch (target branch), leaving the source branch intact. The target branch can then be sync'ed freely to the new parent without affecting the source branch. If the source branch is the parent of the target branch, the port is done using a special procedure called a sync. Refer to section E.9 “Syncing”.

E.6.1 Why Do Porting?

Features developed on the source branch can be made available on a more recent baseline or a totally different baseline. Porting enables feature releases on a more recent codebase that includes other features not in the original baseline, while preserving the ability to do maintenance releases of the earlier version.

E.6.1.1 Porting Considerations and Requirements

The changes to be ported and the target branch must be known prior to porting. In order for the port to be successful, all of the source branch's dependencies should exist in the new parent branch. If both the source branch and target branch are going to be committed to the same parent in the future, there will likely be a lot of merge conflicts and incorrect merges in the code. For other alternatives, contact src-mgmt-req@cisco.com. It is a good idea to get a commitment from the various groups that made the changes on the source branch to help out with merge conflicts before porting.

E.6.1.2 Porting Advantages/Disadvantages

Advantages: Features based on the older baseline can be maintained separately from the new baseline. Porting is sometimes cleaner than a reparent/sync. One would have more control over the changes being merged. Ports are very successful when the changes being ported are quite self-contained and not very dependent on the parent's code. Plan to pick up additional changes/fixes from the source branch beyond the porting point.

Disadvantages: Porting the same feature into several baselines can add to the cost of code maintenance. This is especially true if multiple copies of the feature are floating around. The cost includes the effort in updating each of the "copies" with the bug fixes made on the source branch, and extra testing and release efforts. It is also difficult to track what needs to be double committed from the source branch. Some of this may be avoided through better planning of the release and commit schedules.

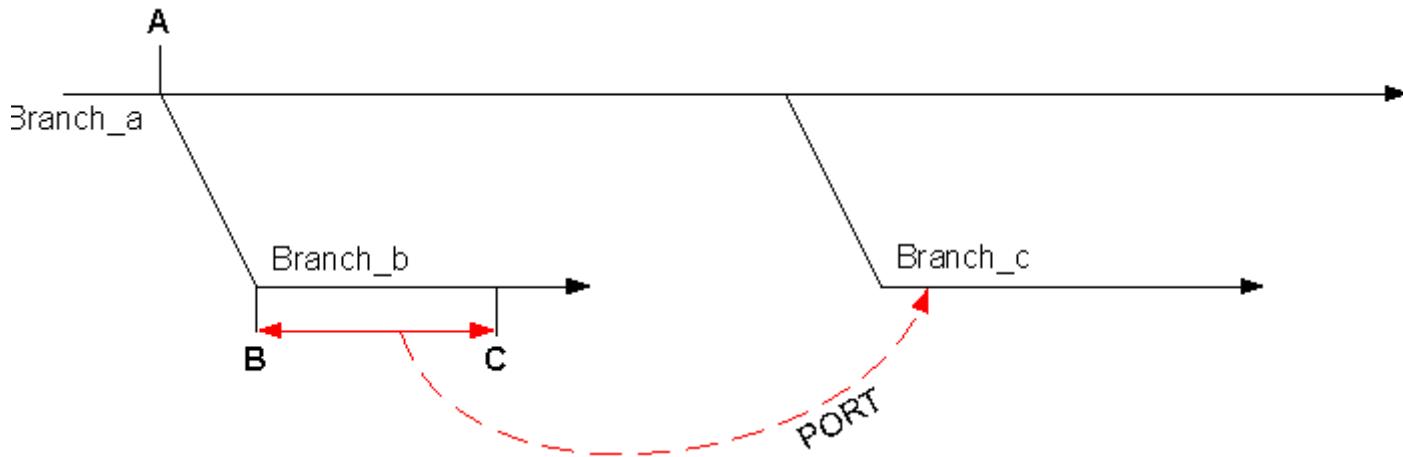
For smaller ports, **diff** and **patch** work well. By using **findmerge**, you are depending on Clearcase to determine what needs to be ported. Using **diff** and **patch**, you have a lot more control over what exactly you want to port.

E.6.2 Porting Applicability

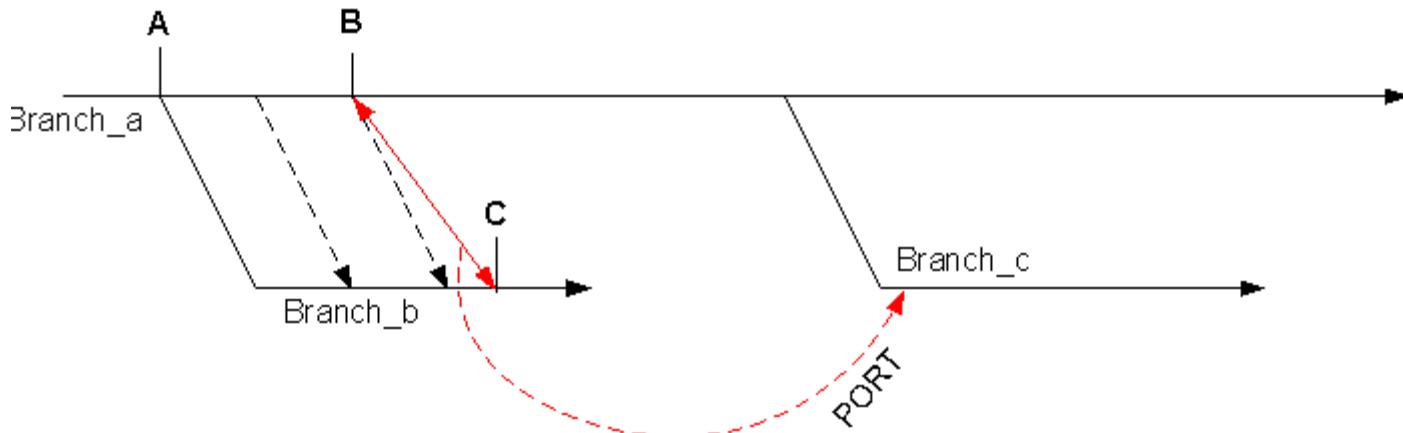
There are different ways of porting depending on the content of the source branch. Understanding the content of the source branch, and deciphering what needs to be included/excluded from the port, is critical.

Some porting scenarios are shown below.

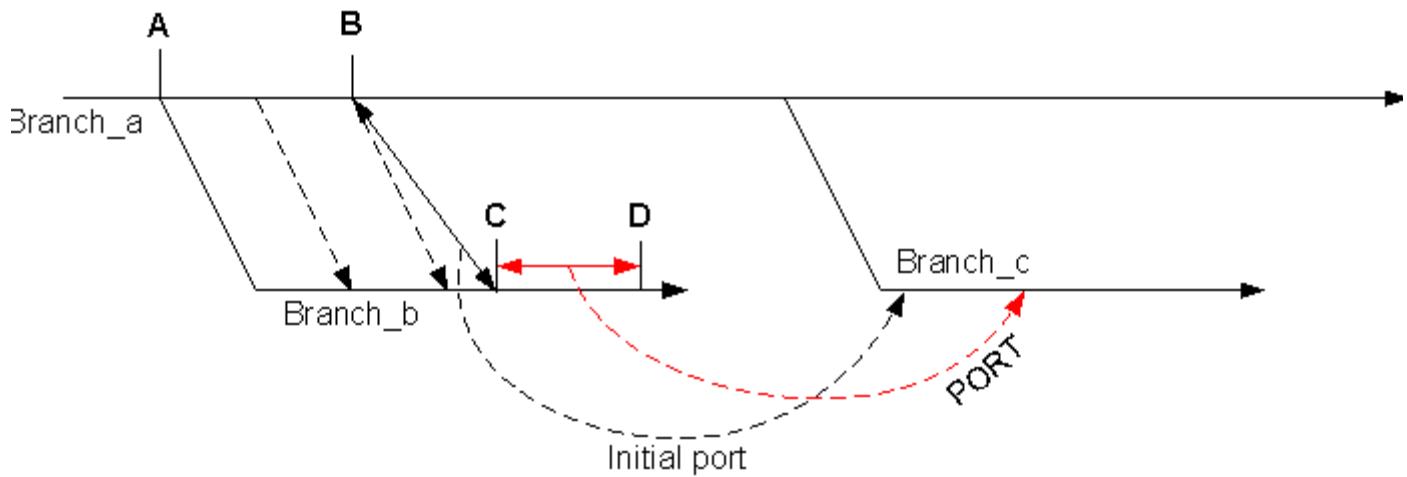
Scenario 1: Figure E-12 shows a port between two releases that have the same baselines. All the dependencies required for the port exist in the target branch. In this case, all changes made on Branch_b from label B to label C are ported to Branch_c.

Figure E-12 A Port between 2 Maintenance Releases with the Same Baseline

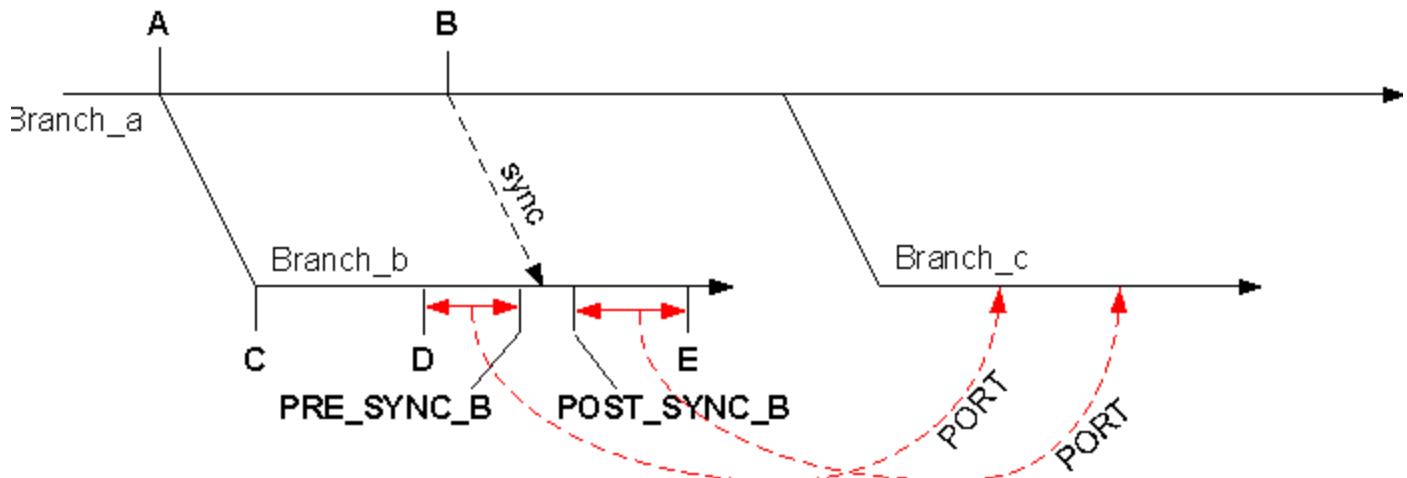
Scenario 2: Figure E-13 shows an example where syncs were performed on Branch_b. Changes made by a sync commit are from its parent and should be excluded from the port. In this case, the delta between B, branch_b's sync point at label B, and C is ported to Branch_c.

Figure E-13 Syncs Performed on Branch Branch_b

Scenario 3: Figure E-14 shows a port between labels C and D, assuming that all changes prior to C were previously ported to Branch_c and that there were no sync changes committed to Branch_b between C and D. In this case, the delta between C and D should be ported to Branch_c.

Figure E-14 Changes Ported to Branch_c

Scenario 4: Figure E-15 shows a port of the changes between labels D and E, assuming that all changes from C to D were previously ported to Branch_c, and that there was a sync committed to branch_b between D and E. PRE_SYNC_B label was placed right before this sync commit and POSTSYNC_B label was placed right after the sync commit. Again, sync changes should be excluded from the port. In this case, the port can be divided into two steps. First, the delta between D and the presync label, PRE_SYNC_B, is ported. Second, the delta between the postsync commit label, POST_SYNC_B, and E is ported.

Figure E-15 A Sync between the 2 Port Points B and C

E.7 Cloning

This section discusses the concepts behind cloning, its applicability, and other issues surrounding it.

E.7.1 What Is Cloning?

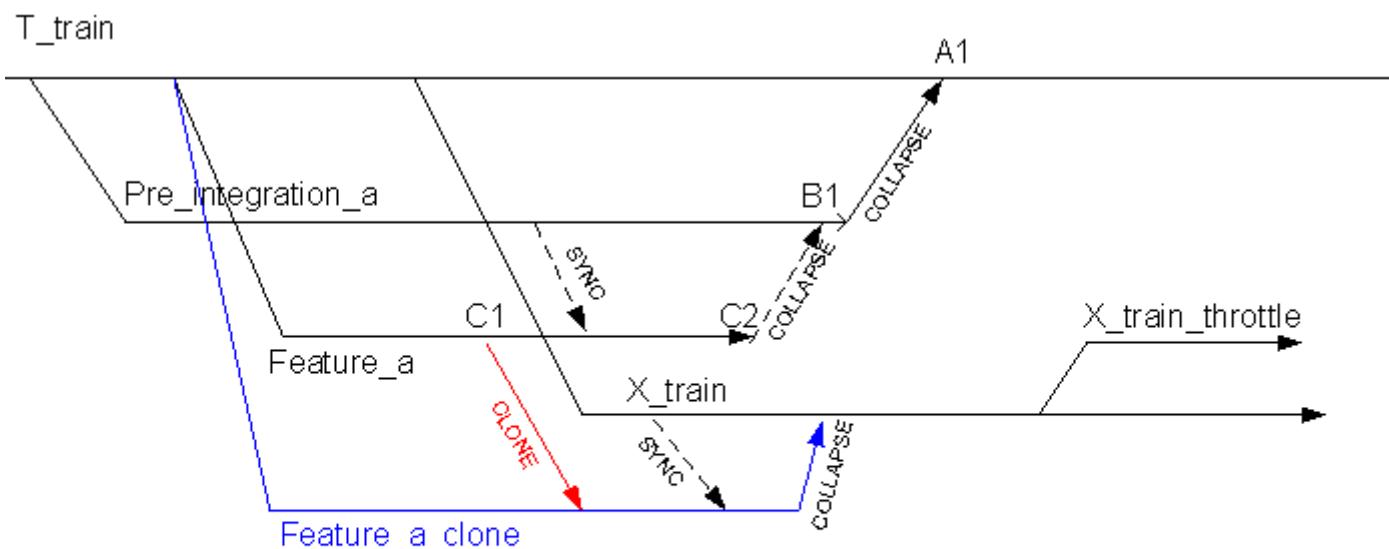
Cloning is the process by which an exact replica of a branch is created. The cloned branch, i.e. the branch created by the cloning process, will have the same parent branch, REF_LABEL, and content as the original branch. Usually the cloned branch is then reparented to a different branch. The results achieved by cloning can also be achieved by performing ports. In most cases where cloning is possible, it is the safer and better alternative to porting.

E.7.2 Cloning Scenarios

There are very few scenarios where cloning is the best option to choose.

Scenario 1: Cloning is done to release a feature from a baseline which will not be updated with the original. This is the most common application for cloning a branch. Figure E-16 is an example:

Figure E-16 Cloning Scenario 1



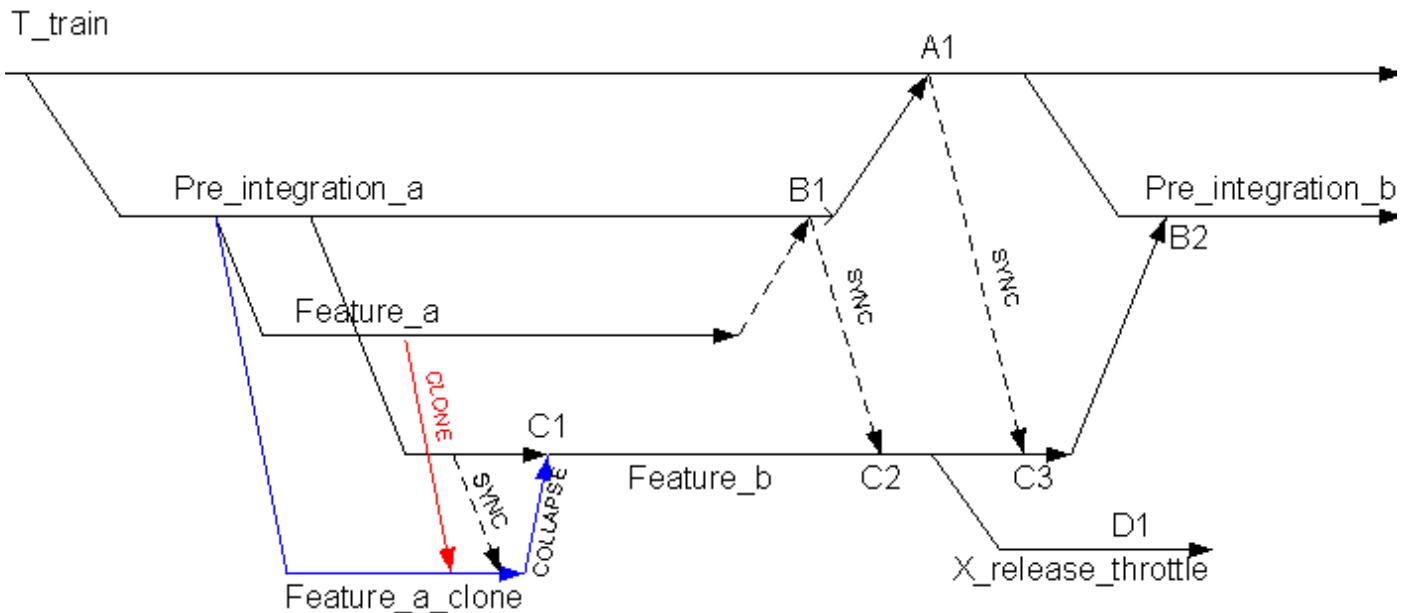
The blue line (“Feature_a_clone” from “T_train” that collapses into “X_train”) shows the branch being pulled, and the red line (“CLONE” from “C1” to “Feature_a_clone”) shows the cloning operation.

In the above case, the feature A would be released to the T train, via the Pre_integration_a branch. However, there was also a need for the same feature to be released off of a special release branch, the X_train. Hence, the Feature_a branch is cloned to Feature_a_clone and then the reparented to the X_train branch. Eventually, Feature A is released off of X_train and T_train.

To ensure successful cloning, all changes and bug fixes made on the original branch (Feature_a) after the clone point, C1, need to be double committed to the cloned branch too. If the cloned branch has been collapsed, the updates will have to be done to the X_train branch. This is a cost of the cloning process (not shown in the diagram).

Scenario 2. Cloning is done to pre-integrate a feature with a different feature so that it can be released to the customers earlier. Figure E-17 is an example:

Figure E-17 Cloning Scenario 2



As per plan, feature A and feature B will be integrated together at point B2 and will be available to be released to the customer at any point in time after that. However, there is a real customer that needs to have the two features, A and B, integrated together and released at an earlier point in time, D1. One way to facilitate that is to create the branch Feature_a_clone by cloning Feature_a and reparenting that to the branch Feature_b. The two features can now be integrated and released from the X_release_throttle.

The same result could also be achieved by porting the branch Feature_a. However, in most cases, cloning is the better option. When Feature_b branch finally collapses to Pre_integration_b, there will be collisions in the code for Feature_a. In most of the cases, this collision is unavoidable and sufficient time should be factored in for that sync and collapse process. Also, the changes made on the original branch, Feature_a, after the clone point, should be recreated on Feature_b to make sure that the released code works well.

E.7.3 Disadvantages of Cloning

Typically, cloning causes more problems than it solves. Hence, it should be used very sparingly. In most situations, there are better alternatives to this. Most of the problems arise from the fact that the original code and cloned code will collide, thus creating a lot of merge conflicts and errors during syncs and collapses. Before proceeding, please send an inquiry to src-mgmt-req@cisco.com, discuss the matter with a sync engineer, and make sure that you exhaust all other possibilities.

E.8 Collapsing

Collapsing a child branch to its parent branch is the process of delivering all the code developed in the child branch (as part of new features and bug fixes) to its parent branch. Conceptually this process is similar to doing a commit of the changes made in a task branch.

A terminal sync is performed first, so that all the code changes made to the parent branch get merged into the integration branch. When an integration branch collapses to its parent, it indicates that

- all the development on the branch is complete
- any bug fixes for the branch need to be committed to the parent branch now
- all the existing task views based on this integration branch cannot be used to commit code into it
- all the active child branches for this integration branch need to be reparented. (typically to the integration branch's parent)

After a collapse, it is possible to pull a view off of a label on the collapsed branch and build images for testing and debugging, but the branch is locked for future commits after the collapse. Different parent branches have their own set of collapse criteria as set by the Program Manager. For example, collapse criteria might include sanity test requirements and resolution of all outstanding severity 1 and severity 2 DDTs before the collapse. Typically a combination of Clearcase and CC tools wrappers are used for the collapse process.

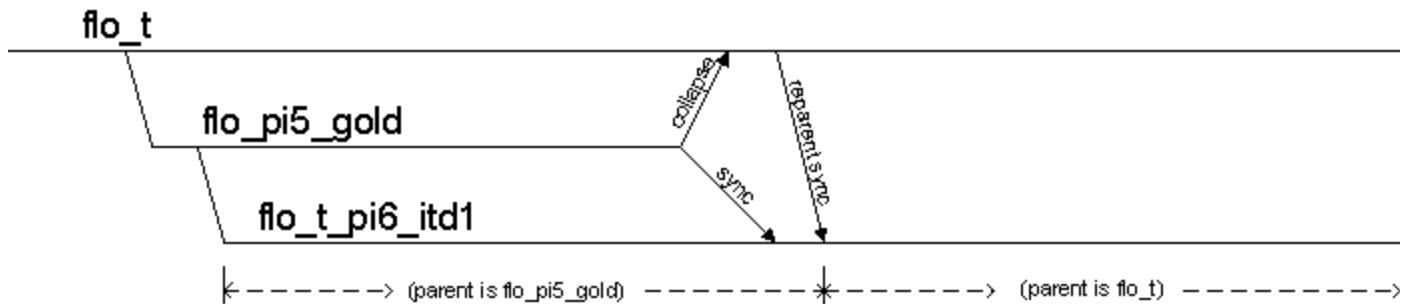
E.8.1 Why Collapse a Branch?

The process of collapsing an integration branch delivers all the code changes done on it—new features, bug fixes, enhancements—to the parent branch for subsequent release.

E.8.2 Assumptions

The integration branch that needs to be collapsed should have had periodic incremental syncs to its parent branch. A terminal sync needs to be done prior to the integration branch's collapse to its parent.

Figure E-18 Collapsing a Branch



In Figure E-18, the child branch has an incremental sync to the label ABC on the parent branch. It has a terminal sync to label PARENT_PRE_COLLAPSE_CHILD on the parent branch. It is then collapsed to the parent branch. An END_OF_CHILD label is laid on the child branch. A PARENT_POST_COLLAPSE_CHILD label is laid on the parent branch. The branches under the child branch should be reparented to the parent branch after the collapse of the child branch.

E.8.3 Different Ways of Collapsing a Branch

There are 2 different ways of collapsing an integration branch: by using a task branch and by using an integration view.

Collapse by converting into task branch: The branch to be collapsed is temporarily converted into a task branch. This task branch is then committed to the parent branch.

Collapse using an integration view: In this case, a "branch collapse" view is created on the parent and the changes made on child branch are merged into the parent using the view.

E.8.3.1 Collapse by Converting to Task Branch

- When to use this method
 - When the integration branch is relatively small (less than 1000 branched elements).
 - When the integration branch does not have any child branches that need to be reparented.
- Advantage
 - The parent branch doesn't need to be locked (although this is recommended for a smoother collapse).
 - The process is quite straight forward.
- Disadvantage
 - The process can be quite slow, if the number of branched elements is high.
 - The process causes spurious merges for its child branches.

E.8.3.2 Collapse Using an Integration View

- When to use this method
 - When the integration branch has other integration child branches that need to be reparented after the collapse.
 - When the integration branch has 1000+ branched elements.
- Advantage
 - The process prevents spurious merges for its child branches.
 - The process is much faster than converting to a task branch, especially for very large integration branches.
- Disadvantage
 - The parent branch has to be locked.
 - You need to have access to parent's view server.

See Table E-1 for a comparison of the collapse processes.

Table E-1 Comparison of Collapse Processes

Feature	Collapse Using Integration View	Collapse Using Task Branch
Impact on parent's schedule	High. The parent branch needs to be locked throughout the terminal sync and collapse	Medium. Same as any task view commit.
Requirement that parent branch be locked	Yes. The parent branch has to be locked to use this process	No. The parent branch need not be locked. However locking the parent branch is safer and more desirable.
View server used for collapsing the branch	Parent branch's server. The collapse is done by creating a view based on the parent branch. This is done on the view server where the parent branch is mastered. In other words, for the “convert-to-task” method, we have to use the server where the child is mastered, and for the “use-integ-view” method, we have to use the server where the parent is mastered.	Child branch's server. The collapse is done using a view based on the child branch. This is done on the view server where the child branch is mastered
Ease of reparenting child branches of the branch being collapsed	Relatively easy. As we know that the code on the parent branch has been replaced, steps can be taken to ensure that syncs after the reparent will use the right base version for files. This prevents spurious merges from occurring.	Tricky. The code on the parent branch may not be identical to the child branch. This may pose difficulties in reparenting the child branches of the collapsed branch, causing spurious merges.
When recommended	This is recommended when the branch being collapsed has child branches that need to be reparented.	This is recommended for small integration branches and in cases where we do not have access to the view server in which the parent branch is mastered.

E.9 Syncing

Syncing is the process of merging the code changes in a parent branch to its child branch. A sync is usually done several times during the lifetime of an integration branch. Typically it involves bringing in all the code changes in the parent branch that come from new features/ featurettes or bug fixes committed to it since the last sync (or since the child branch was created, if this is the first sync for the child branch). It also moves the reference label on the child branch to the fixed, full label on the parent branch to which the child branch is sync'd.

E.9.1 Why Do Syncs?

During a sync, only the files that have been modified on the parent branch and have a version on the child branch are merged. (Those code changes in the child branch that are different from the corresponding parallel changes committed to the parent branch are marked as conflicts). These conflicts need to be resolved appropriately to pick the right code for the child branch. There are three types of sync that are typically done during the life cycle of an integration branch: incremental, terminal, and reparent.

E.9.1.1 Incremental Sync

The incremental sync brings in the code changes made on the parent branch since the previous reference label (i.e. the previous sync point) into the child branch. The code changes are typically bug fixes and new features/featurettes.

E.9.1.2 Terminal Sync

The terminal sync is the last sync done on a child branch to the latest stable label on its parent branch before it collapses into the parent branch. This sync is followed by the collapse process to commit all the code development done on the child branch into its parent branch.

E.9.1.3 Reparent Sync

The reparent sync is done when an integration branch which is branched off of one parent branch is moved under a different parent branch (reparented). Here, as the last step in the reparent process, the child branch is sync'd to a suitable stable label on its new parent. This sync ensures that the child branch can get the code from the new parent branch.

E.9.1.4 Sync Assumptions

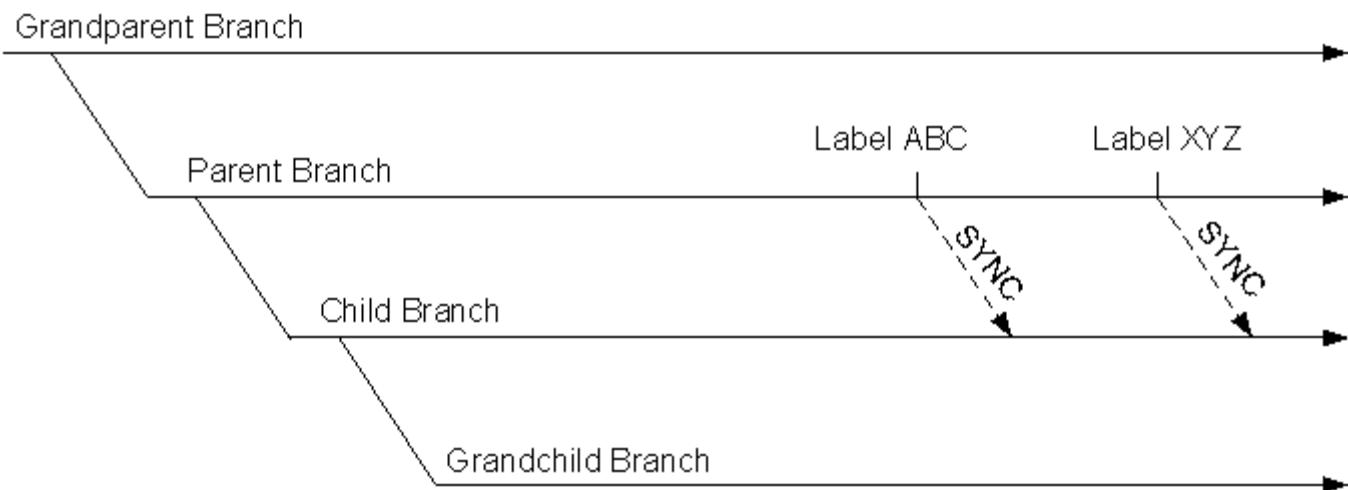
In Cisco's distributed ClearCase environment, there are many VOB replicas. Each integration branch is mastered on one of these VOB replicas. The sync process must be executed on a client machine with the same VOB replica as the one that masters the integration branch.

The DDTs_STATE attribute of the child branch is set to MER to effectively lock out all code commits except those brought by the sync. This method is better than placing actual ClearCase locks on the branch because it allows engineers to checkout files from the branch while the sync process is being done.

For Cisco IOS syncs, the Cleartool **findmerge** and **rccsmerge** are used to merge the code diffs from the parent branch to the child branch.

E.9.2 What is a Sync?

Figure E-19 Branch Hierarchy



In Figure E-19, the child branch has been sync'd up to the Label ABC on its parent branch. Therefore its REF_LABEL was set to Label ABC. When the sync is performed on the child branch to Label XYZ, the code diffs between these two labels on the parent branch need to be merged into the child branch. The following subsections let us look at these changes that occur during the sync.

E.9.3 Elements with Child Versions

The elements that get modified on the child branch have a version on the child branch. For example, the flo_isp branch was pulled off of florida at Label V122_0_5. After the branch is pulled, the element version seen on the flo_isp branch for the file network.c is /vob/ios/sys1/sys/if/network.c@@@/main/florida/12. When this file gets modified on the child branch, then the child version, network.c@@@/main/florida/flo_isp/1 is created. Further versions /2, /3 etc. get created as more changes to the file get committed to the child branch. Staying with Figure E-18, during the sync, the code changes between the labels ABC and XYZ on the parent branch need to get merged in all the elements with child versions. For each of these elements, the **findmerge+rcsmerge** tool will bring in all the code changes made on the parent between labels ABC and XYZ and merge them with the versions seen on the child branch. If the code changes brought in from the parent conflict with the parallel code changes made in the child version, then these will be flagged in the sync view. These conflicts need to be manually resolved as appropriate. If there are no parallel code change conflicts between the parent and child branch versions of the elements, then the code gets merged automatically. All these child-versioned elements with merged code are checked out by the **findmerge** tool. When the sync gets committed, these elements get checked into the child branch.

E.9.4 Bleed-through Files

These are the files that have not been modified in the child branch. So, the versions seen in the child branch for these files are those from the parent branch. These files are not affected by the **findmerge** process and so are not part of the sync change set to commit in the child branch. Typically the number of bleed-through files is much greater than the number of child versioned files. So, the

changes introduced by these are a lot more numerous than the changes committed during a sync to the child branch. This is especially true for branches with many ancestors, like the grandchild branch in Figure E-19.

Let us assume that the parent branch has 1000 files that are modified between the labels ABC and XYZ. If say, 30 of these files are modified in the child branch, then the code diffs for these 30 files between labels ABC and XYZ is merged into the child branch. The remaining 970 are bleed-through files.

Initially when the child branch is sync'd to label ABC, the child branch views will see parent branch file/element version corresponding to label ABC. When a sync view is created, the config spec is updated to see all the parent branch files/elements corresponding to label XYZ. When the sync gets committed, all these 30 file changes will be checked into the child branch.

E.9.5 Backing Out Some Merged Code

Sometimes, the code changes from a parent branch that are associated with a specific DDTs should not be sync'd into the child branch. There can be various reasons for this.

- The bug fix could have been committed in both the child and parent branches because the child branch needed an early fix.
- The Code changes for the bug fix in the parent branch may not be applicable on the child branch or the code can be totally different on the two branches for a feature/functionality.

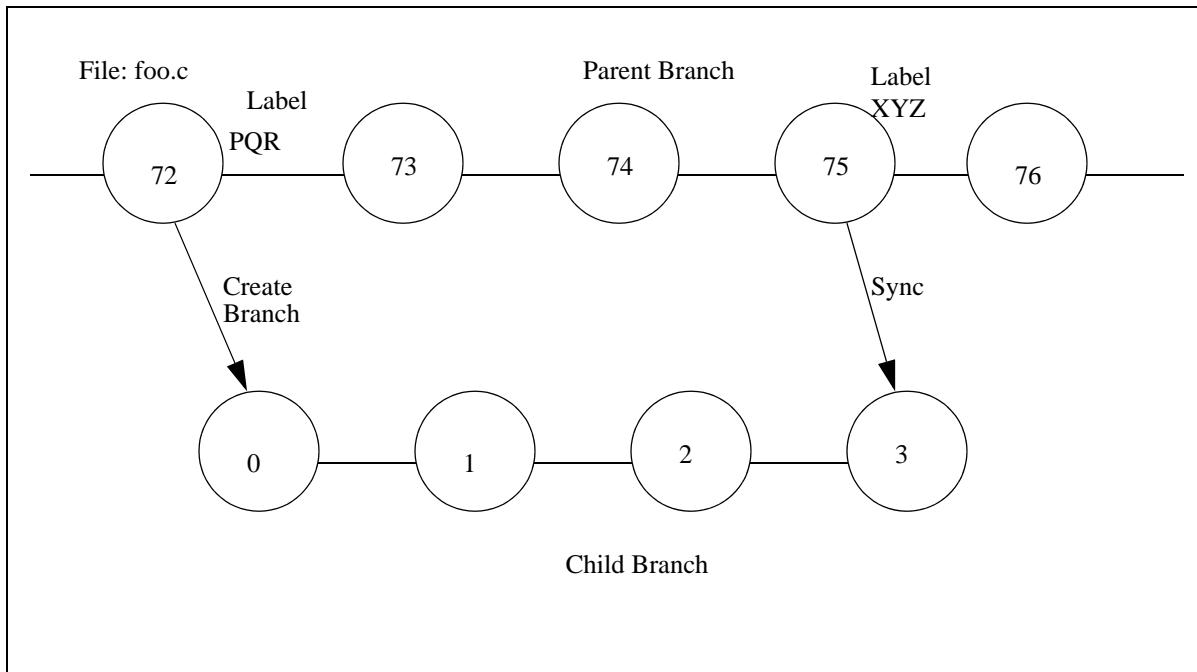
The sync engineer needs to handle these DDTs commits appropriately.

E.9.6 Sync Advantages/Disadvantages

With new features and bug fixes continually committed to a parent branch, the code between the parent and child branches starts diverging. With periodic syncs, the child branch picks up the parent code changes. The frequency of doing syncs needs to be chosen optimally as suited for the branch. Doing very frequent syncs will result in introducing parent branch code instability in the child branch. On the other hand, if syncs are not done for a long period of time, it can result in a very large code change set with too many conflicts.

Syncing can only be done between a child/parent branch and not between any arbitrary branches or arbitrary sync point. For example, to get new code changes made in a grandparent branch (not shown) in Figure E-20, first the parent branch needs to sync to a suitable label on the grandparent branch. Then, the child branch needs to sync to a suitable label on the parent branch that was laid after the parent branch sync.

Figure E-20 File foo.c Changes with a Sync

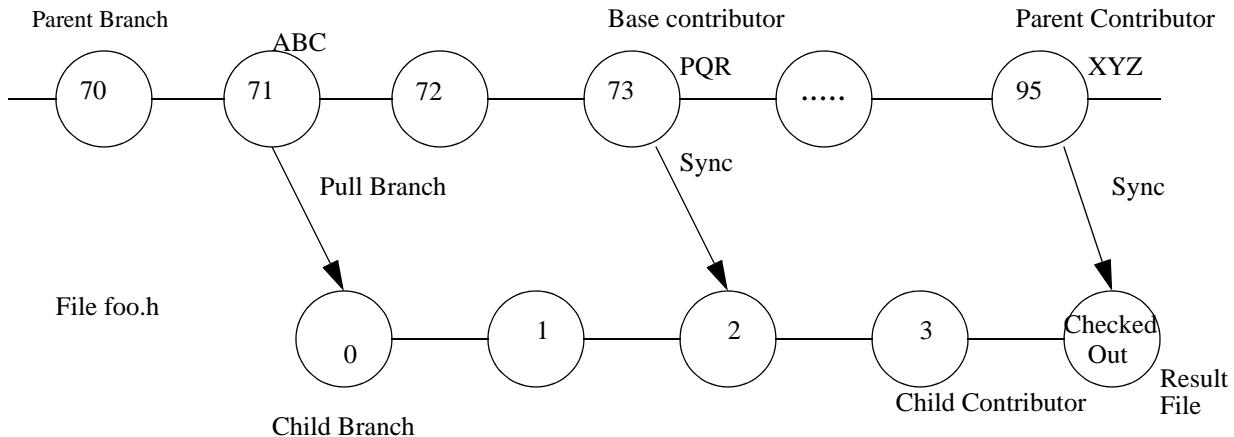


The 0 is a “non-version” that is created by ClearCase. Actually, the 0 version is the same as the previous version (i.e. 72 on the parent).

In Figure E-20, when foo.c is committed to the child branch, the file element is branched in Clearcase and the modified file becomes version 1 of the file element on the child branch. Another modification and commit through a bug fix to the child branch creates version 2. Similarly, code commits made to foo.c in the parent branch creates versions 73, 74, 75, etc. In the figure above the circles represent the versions of foo.c in the parent and child branches. The file foo.c is branched at version 72 of the parent branch and the first modifications to it are committed as version 1 on the child branch. Version 0 on the child branch is identical to version 72 on the parent branch. The child branch was pulled off of label PQR on the parent branch. This label corresponds to version 72 of foo.c. Clearcase determines this data based on the child branch configuration specification, created during the child branch creation.

As you can see, label XYZ on the parent branch corresponds to version 75 of foo.c. A sync is done for the child branch to label XYZ. The changes to foo.c between the parent branch versions 72 and 75 get merged and are committed as version 3 on the child branch. The REF_LABEL gets updated to XYZ, signifying that the changes up to version 75 on the parent have been sync'd to the parent. The next sync to a later label will consider the diffs from version 75 to the corresponding newer version on the parent branch.

The versions of a file that are used to do a merge are referred to as “contributors”. In this example, version 75 of the parent branch and version 2 of the child branch are called the parent contributor and child contributor, respectively. To isolate branch specific changes, we have a third contributor called the base contributor or common ancestor. It is the most recent version of the file held in common between the parent and child contributors of the merge. The Clearcase **findmerge** tool helps in determining the base contributor.

Figure E-21 File foo.h Changes with a Sync

In Figure E-21, the child branch is pulled off of label ABC. The child branch is sync'd to label PQR and then to label XYZ. For the sync to label XYZ for the file `foo.h`, the parent contributor is version 95, the child contributor is version 3 and the base contributor is version 73. The code difference between the base contributor and each of the parent and child contributors specify how the file has diverged since the last sync. If there are overlapping changes, these 2 code diffs determine those and assist in resolution of conflicts due to overlapping changes. After the contributors are merged, a new version 4 will be committed to the child branch. If during the sync the child branch is kept unlocked for engineers to commit additional changes during the sync, then the sync engineer needs to integrate these new changes prior to the sync commit. The version number on the child then will also change to the next available number.

In the sample output below, for the florida branch, the label V122_16_13 refers to version 108 of file `/vob/ios.sys1/sys/if/network.c`. The label V122_17_1U refers to version 109 of the same file. When the child branch `flo_isp` with a current REF_LABEL of V122_16_13 (i.e. sync'd up to this label) is sync'd to V122_17_1U, the code diffs between these versions get merged and committed to `flo_isp` for `network.c`. The REF_LABEL is also updated to V122_17_1U:

```

villa:41> cc_find -l V122_17_1U /vob/ios.sys1/sys/if/network.c
/vview/SYN-flo_isp.V122_17_1U/vob/ios.sys1/sys/if/network.c@@/main/florida/109
villa:42> cc_find -l V122_16_13 /vob/ios.sys1/sys/if/network.c
/vview/SYN-flo_isp.V122_17_1U/vob/ios.sys1/sys/if/network.c@@/main/florida/108
villa:44> ct diff -pre -ser /vob/ios.sys1/sys/if/network.c@@/main/florida/109
*****
<<< file 1: /vob/ios.sys1/sys/if/network.c@@/main/florida/108
>>> file 2: /vob/ios.sys1/sys/if/network.c@@/main/florida/109
*****

```

The Clearcase **findmerge** command is a combination of a **find** and **merge** command. It searches for elements that require a merge from a specified version to the version in the current view. Optionally, **findmerge** can recursively traverse one or more VOBs in the current view and merge all the relevant files, checking out the file elements as necessary. For Cisco IOS syncs, Clearcase **findmerge** is used to identify the elements that require a merge and **rcsmerge** is used to do the 3-way filemerge. This is done by passing the **cc_merge -r** parameter to the **-exec** option in the Clearcase **findmerge** command. The resulting merged file is created in the sync view. This checked out merged file gets committed to the child branch after conflict resolution, building, and testing.

E.10 Roles and Responsibilities of the Stake Holders during a Sync

The purpose of this section is to give a brief description of the roles and responsibilities of all the stake holders involved during the sync process.

E.10.1 Stake Holders Involved

The stake holders involved in the sync are:

- Release Program Manager (PM)
- Sync Engineer—the person performing the sync. For development branches, it is usually an engineer who does this. In the case of release branches, it is typically an engineer whose main role is to perform syncs.
- Development Manager
- Development Engineers
- Dev Test Engineers

Note This is to be used only as a guideline. The roles and responsibilities of the stake holders involved may vary depending on the requirements of various branches.

E.10.1.1 Release Program Manager (PM)

This stake holder is expected to:

- Schedule the syncs.
- Initiate the sync request. The request is made to the designated sync engineer. There is no set standard on how this has to be done. Different groups have different protocols.
- Find the resources to perform the sync.
- Provide the label for the sync. This is provided to the sync engineer. There are many ways in which the sync label is chosen. It is difficult and lengthy to enumerate them all here.
- Decide the timing for the sync.
- Provide the list of images that need to be built to verify the correctness of the sync.
- Be the point of contact for the sync engineer for any sync-related issues.
- Be responsible for escalating sync-related issues with various teams.

E.10.1.2 Sync Engineer

This stake holder is expected to:

- Perform the sync.
- Figure out the server/partition on which to perform the sync
- Resolve conflicts to the best of their ability.
- Assign unresolved conflicts to be resolved by DE's to the best of their ability.
- Provide all necessary information required to resolve the conflicts and fix the errors for the DEs.

- Provide all the necessary information required to resolve the conflicts, review conflict resolution, and fix the compilation errors for the DEs.
- Build the relevant images and fix build errors to the best of their ability.
- Send out the sync diffs to the DE groups for review to make sure that changes are sync'd correctly.
- Communicate the sync status to all the people concerned. Typically, the communication is done via email, however, there are plans to move some of the communication to “updating a web page.”
- Provide images for sanity testing to the testers, if applicable.
- Address any sync-related queries from various teams.
- Communicate to the PM if escalation is required.
- Set the time expectations at various stages during the sync.

The above responsibilities may vary depending on whether it is a development branch sync or a PI/premerge branch sync. (PI is a Pre-Integration branch. These syncs are done by the Central Sync Team, not developers).

A development branch sync is typically done by a DE, who takes on the responsibility of resolving conflicts, fixing errors, and getting other DEs involved as needed.

A PI/premerge branch sync is generally done by a sync engineer from the Central Sync Team. His/her role is to drive the sync. The sync engineer also works on the conflicts/build errors, depending on bandwidth.

E.10.1.3 Development Manager

This stake holder is expected to:

- Make sure the DEs treat sync issues as high priorities.
- Make sure all sync conflicts/errors have owners assigned.
- Help the PM find the server/partition for the sync.

E.10.1.4 Development Engineers

These stake holders are expected to:

- Have a good understanding of what happens during a sync, what changes get merged, why conflicts emerge, and how to resolve them correctly.
- Resolve conflicts and compilation errors sent out by the sync engineer in a timely manner.
- Review the sync diffs to make sure the sync changes are good.
- Send manually resolved files out for review.

E.10.1.5 Dev Test Engineer

This stake holder is expected to:

- Be available to sanity test the sync images if required by the team.
- Perform the sanity test and communicate the results in a timely manner.

Doxxygen Instructions

Added a note related to doxygenated documents' location for the APIs in first page of the chapter.
(May 2009)

Completely revised chapter (replaced docx information). (March 2009)

Note Currently, the Doxygenated documents for all the APIs that are used by the programmers is stored at the /auto/ios_comp/api-docs location. Therefore, you can generate the API docs to the specified location and point folks to it. After the completion of the automated process, all guides will point to the official generated content.

F.1 What Is Doxygen?

Doxxygen is an open-source tool that generates documentation from source code. It is the most widely used documentation generation tool in the industry, representing millions of lines of code. Most open-source projects use it, including Mozilla, GNU, Apache, MySQL, KDE, and XFree86.

It is also used at Cisco in several BUs (NSSTG, ONG (Optical), CCBU, VTG, EMSBU, EVVBU, NMTG, MFG, and more).

F.2 Why Has Doxygen Been Selected at Cisco?

It has been selected in preference to DocX (which is used in IOX source code, for example) for the following reasons:

- Ease of adoption
- Small development overhead
- Ease of use
- Quality of generated pages
- Extensive keyword hyperlinking
- Extensive cross-references
- Customizable look and feel
- Optional source code inclusion (great for code reviews)
- Robustness

- Wider industry adoption
- Active tool updates and support
- Extensibility
- Output formats (such as HTML, XML, PDF, and RTF)
- Extensive language support (such as C, C++, Java, C#, Python, and Ruby)

F.3 What Is the run-doxygen Script?

The term **run-doxygen** refers to the **run-doxygen** script and to the production environment that is associated with the script. The **run-doxygen** script is a wrapper around the open-source Doxygen tool. Its main purpose is to provide a simple command-line interface for producing Doxygen documents at Cisco so that:

- All Cisco Doxygen documents have the same look and feel.
- Cisco-specific Doxygen tags can be readily used.
- Developers working on the same project can produce documents without interfering with each other.
- Documents are generated in a directory that has a layout consistent across projects.
- API documentation for software components that are released can be archived properly in a consistent way.

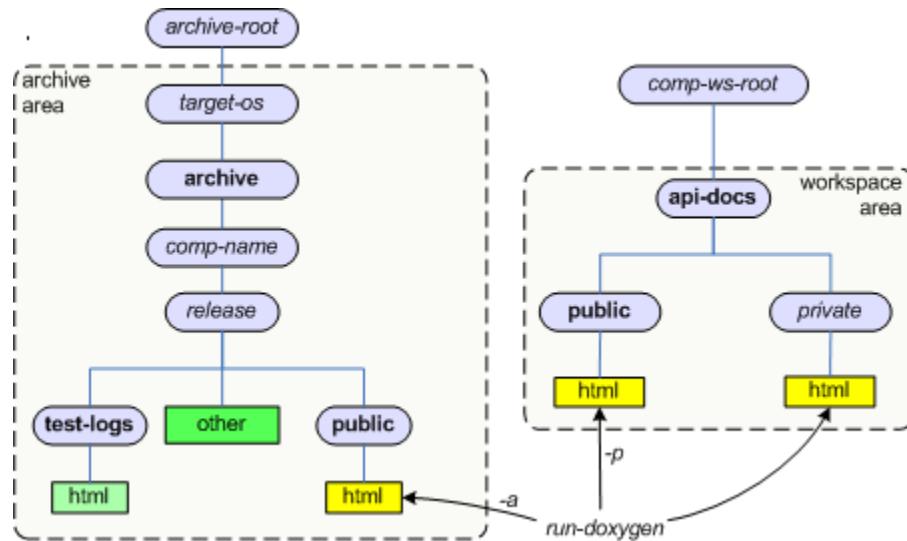
Although **run-doxygen** was initially developed in the context of the Cross-OS and IOS Componentization projects, it can also be used for any other Cisco software project that uses C/C++ as the main programming language.

run-doxygen currently has the capability to generate documentation on Solaris and Linux systems in HTML format. PDF format is available *only* on Linux systems; Solaris has unresolved PDF tools issues. Doxygen documentation is intended to be viewed mostly in HTML format; PDF versions are used mainly as an efficient way to store the documentation in EDCS when required.

F.4 Workspace Area Versus Archive Area

Figure F-1 illustrates the workspace area versus the archive area.

Figure F-1 Workspace Area Versus Archive Area



run-doxygen defines two types of production:

- Day-to-day documentation production that is stored in a *workspace area*. These documents are generally used for reviews and are deleted when the development task is completed.
- Archived documentation production. These documents are archived for each version of the software in a space called the *archive area*.

Prepublication and publication are combined in the day-to-day activities that happen in the workspace area (engineers publish/generate Doxygen code during their day-to-day activities). The archive is a library of past day-to-day activities and is a repository for the engineers so they can find Doxygenated code that has already been published.

F.4.1 Workspace Area

The workspace area contains only Doxygen documentation generated from the source code.

The workspace area has the following folders:

- **public**—the public folder for the component, typically contains the latest stable version of the API documentation
- **private**—the private folder that contains private versions of the API documentation. Each developer can generate his or her own version of the component documentation, based on his or her own view of the source code.

F.4.2 Archive Area

There are plans to provide a unique centralized archive area for all software components at Cisco. Until this is finalized, **run-doxygen** can be configured to use any archive area.

The archive area has the following levels:

- *target-os* contains the archive documentation for all software components that will run under the same target OS. Examples of target-os are **ios** (all components that run on IOS), **iox** (all components that run on IOX), and **xos** (all components that run in the XOS environment).
- *comp_name* is the component name (for example, **xoslib**).
- *release* is the archived release number. The unit test result log files (if available) can be stored in the same location as the archive.
- *public* contains the Doxygen documentation for the component.
- *other* represents other deliverables that need to be archived (for example, UNIX package).

F.5 Setting Up the Environment

A Doxygen run requires 4 files:

- The **doxygen** binary itself
- The **run-doxygen** script (which is a shell script file)
- The Cisco Doxygen global config file (**doxygen.cfg**)
- A project-specific **.run-doxygen** config file

doxygen is located in `/router/bin`.

The **run-doxygen** scripts and the **doxygen.cfg** global config file are located in `/auto/ios_comp/tools/bin`.

The template for the **.run-doxygen** configuration file can be generated using **run-doxygen** with the **-T** option.

F.5.1 Instructions to Set Up run-doxygen for Your Project

Step 1 Get the template of the **.run-doxygen** configuration file using **run-doxygen -T** and save it to a name describing your own Doxygen configuration. For example:

run-doxygen -T > bgp.run-doxygen

The Doxygen configuration filename must terminate with the **.run-doxygen** extension.

Step 2 Edit that copy and enter information specific to your project (the comments in the default file provide a description of each variable).

Step 3 You could optionally create your own **run-doxygen** wrapper that will invoke the **run-doxygen** script and pass the command-line argument to pick your version of the **.run-doxygen** configuration file.

F.5.2 Special Notes for Remote Sites

A normal **run-doxygen** invocation should take a few seconds to complete for small- to medium-size APIs (fewer than 40 header files). It may take a few minutes for larger API sets.

If you are not located on the SJ campus, you might experience very slow processing time when invoking **run-doxygen** (over 10 minutes for large API sets). This is because of 2 factors:

- The Doxygen binaries are located physically in the US.
- The output folder may be located on a remote file server.

To speed up the Doxygen runs, it is recommended that you use a local output folder (specified in the OUTPUT_WORKSPACE variable in your **run-doxygen** configuration file). Make sure that this local storage is accessible through a URL because you will need to browse the HTML output from your browser.

If the performance is still too slow, the other option is to install Doxygen on a local file system (until the correct Doxygen binary is officially deployed).

For any questions, contact doxygen-trolls@cisco.com or xos-internal-dev@cisco.com.

F.6 How to Run Doxygen

Once the project-specific **run-doxygen** configuration file is ready, using **run-doxygen** is straightforward:

% **run-doxygen -h**

The syntax is:

```
run-doxygen [-s] [-a] [-h] [-X] [-c <config-file>] [-f <pdf-file>] [-p]
            [-R <rtf-file>] [-T]
            [-v <api-version>] [-e <edcs-number>]
            [-o <output-dir>] [<folder-name>]
```

The command's arguments are in Table F-1.

Table F-1 Arguments for run-doxygen

Argument	Description
-s	Add all source folders to the Doxygen output. Default: only the public include folders are scanned.
-a	Generate into the archive area. Default: generate into the workspace area.
-h	This help message.
-X	Exclude examples.
-c	Use a specific run-doxygen config file <config-file>. Default: pick up a file named .run-doxygen located in: 1) The current directory as a first choice. 2) The directory where run-doxygen is located otherwise.
-f	Generate a PDF format manual into <pdf-file>.pdf. <pdf-file> can be a relative or absolute filename (do not append the .pdf suffix).
-R	Generate a RTF format manual into <rtf-file>.rtf. <rtf-file> can be a relative or absolute file name (do not append the .rtf suffix).
-p	Use the public folder name (same as <folder-name>="public").
-v	Use a specific API version. Default: pick up the version from the setup file (PROJECT_NUMBER). If -v is set, PROJECT_NUMBER will be forced to <api-version>.

Table F-1 Arguments for run-doxygen (continued)

Argument	Description
-e	Add an EDCS document number to the front page of the PDF file: <edcs-number> EDCS document number (For example, -e EDCS-850120).
-o	Specify an output directory under which the folder <folder-name> will be created. Overrides the OUTPUT_WORKSPACE variable in the config file and the \$DOX_OUTPUT_WORKSPACE environment variable.
-T	Dumps the .run-doxygen configuration template file to stdout.
<folder-name>	Use this specific folder name (overrides -p). Default: the folder name will be set to: <ul style="list-style-type: none"> • The current Clearcase view name if -a is not set. • The PROJECT_NUMBER variable if -a is set.

Environment variables:

- \$DOX_OUTPUT_WORKSPACE overrides OUTPUT_WORKSPACE variable in the config file.
- \$DOX_OUTPUT_WORKSPACE_URL overrides URL_WORKSPACE variable in the config file.

Note The **-f** option is currently supported ONLY on Linux systems and not on Solaris because of a Solaris PDF tools issue. If you require PDF output on a Solaris system, generate the PDF on a Linux system and transfer the PDF file back to the target Solaris system.

F.6.1 Examples of Using the run-doxygen Script

To generate a private version of Doxygen documentation from your local view into the workspace area (this will require a **cc-tools** view since the output folder name will be retrieved from the current Clearcase view name):

% run-doxygen

To generate a private version of Doxygen documentation into the workspace area under the **test-foo** folder:

% run-doxygen test-foo

To generate a private version of Doxygen documentation into the workspace area under the **bgp-test** folder and use a specific version of config file **/project/bgp/run-doxygen.cfg**:

% run-doxygen -c /vob/ios.comp/bgp/docs/bgp.run-doxygen bgp-test

To archive the Doxygen documentation with a version set to **bgp@3.1.0** and use the **bgp** configuration file:

% run-doxygen -a -v bgp@3.1.0 -c /vob/ios.comp/bgp/docs/bgp.run-doxygen

To generate the PDF documentation into a file named **bgp-api-3.1.0.pdf** with a version set to **bgp@3.1.0**, use the **bgp** config file and use a temporary folder named **temp-bgp-pdf**:

% run-doxygen -a -f bgp-api-3.1.0.pdf -v bgp@3.1.0 -c /vob/ios.comp/bgp/docs/bgp.run-doxygen temp-bgp-pdf

F.7 Using the Tags

Doxygen uses tags inserted into comment blocks in the source code or public header files to indicate what content to include in the documentation.

Doxygen supports two ways to identify Doxygen meta-comment blocks and two characters to start all its commands: “@” and “\”.

For consistency’s sake, the Component Framework and Cross-OS APIs should only use the javadoc style:

- Start all Doxygen comment blocks with “/**”.
- Start all Doxygen commands with “@”.

F.7.1 Escaping Special Characters

If you want to use special Doxygen command characters in comment block text, such as “#”, “/”, “\”, “>”, “<”, or “?”, they must be preceded them by a “\” or “@” escape character. Otherwise, Doxygen will try to interpret them and the text that follows as Doxygen commands, resulting in unpredictable output.

F.7.2 Doxygen Comment Blocks

By default, the first sentence (delimited by the first period (.) character) of each comment block is used by Doxygen as the brief description displayed for the corresponding symbol in the symbol summary lists at the beginning of the output results.

Here is an example of a Doxygen comment block for a function that returns a Boolean value and takes two input parameters, `queue` and `entry`:

```
/**
 * Check if a given entry is present in the queue. The entry is present if
 * there is an element in the queue that is equal to entry (using the (void*)
 * == (void*) equality * operation).
 * @param[in] queue A valid pointer to an xos_queue (cannot be null).
 * @param[in] entry The entry to look for in the queue.
 * @return TRUE if the entry is present in the queue, FALSE otherwise
 * @pre The queue must be initialized properly.
 * @post The queue is not modified.
 * @safety #NotThreadSafe
 * @note The cost of this lookup is O(n).
 */
```

Here is another example of a Doxygen comment block for a well-documented API function:

```
/**
 * bitfield_set()
 * To set a bitfield, call the bitfield_set() function.
 *
 * #include "bitlogic.h"
 * void bitfield_set(ulong bitnum, dynamic_bitfield *bitfield);
 *
 * @param[in] bitnum The bitfield number.
 * @param[in] bitfield Pointer to the bitfield to be set.
 * @param[out] None
 *
 * @return None
```

```

*
* @usage
* This function is used to set a bit in a bitfield, and if the bitfield
* is too small, reallocate it. Do not call this function from the interrupt
* level if the bitfield will need to be expanded; dynamic bitfield expansion
* can occur only at process level.
*
* @codeexample
* For example, the bitfield_set() function is used to set poisoned bits in
* the sys/ipmulticast/dvmrp.c file, as shown here:
* @code
* // If route exists or phantom exists, use it. Otherwise, add phantom and
* // get route so we can set the poison bits.
* route = dvmrp_get_route(mvrf, source, mask, FALSE, TRUE);
* if (!route) {
*     dvmrp_add_route(mvrf, source, mask, nbr, idb,
*                      DVMRP_INFINITY*2, 255, TRUE);
*     route = dvmrp_get_route(mvrf, source, mask, FALSE, TRUE);
* } else if (route->phantom) {
*
*     TIMER_START(route->expires, DVMRP_EXPIRATION_TIME);
* }
* if (route) {
*     if (set) {
*
*         bitfield_set(idb->if_number, &route->poisoned);
*         DVMRP_IN_BUGINF(source, ("", poison set %d", idb->if_number));
*
*     } else {
*         bitfield_clear(idb->if_number, &route->poisoned);
*         DVMRP_IN_BUGINF(source, ("", poison clear %d", idb->if_number));
*     }
* }
* @endcode
*
* @see bitfield_check(), bitfield_clear(), bitfield_clear_many(),
*      bitfield_clearmask(), bitfield_destroy(),
bitfield_find_first_clear(),
*      bitfield_find_first_set(), bitfield_lock(), bitfield_set_many(),
*      bitfield_setmask(), bitmask_any_bits_in_first_are_set_in_second()
*/
void bitfield_set(ulong bitnum, dynamic_bitfield *bitfield);

```

F.7.3 Using Tags That Specify Descriptions

When you are using tags that specify a {description}, the content of the description should use the syntax and formatting guidelines. For example, white space lines between lines of text are not translated into the HTML presentation, so all lines would run together unless separated by Doxygen @n, @par, or other commands that force line breaks.

- The Doxygen @n command forces a newline in the output.
- The Doxygen @par command forces greater paragraph spacing within other @par sections, forces new sections to begin, and indents the text paragraphs in the section. These result in better output formatting when text is processed by Doxygen into HTML output.

F.7.4 Descriptive Text Begins on Same Line as Tag

In general, the {description} should start right after the tag and not begin on the next line. For example, the @safety tag requires some text on the same line as the tag. The following usage will cause a liner warning:

```
@safety
```

```
text
```

This is the correct usage:

```
@safety text
```

F.7.5 Alphabetical List of Recommended Doxygen Tags

The complete list of Doxygen tags is defined in:

<http://www.stack.nl/~dimitri/doxygen/commands.html>

which is the official Doxygen website. This URL can be helpful when attempting to troubleshoot Doxygen issues.

Table F-1 defines the subset that is recommended for your use for the Component Framework and Cross-OS APIs (in alphabetical order).

Table F-2 Alphabetical List of Doxygen Tags

Command	Recommended Usage
@code { <i>code</i> }	Delimits the start of a section that contains C code, or example error message, command-line, or output display. Entire block must be terminated with the @endcode tag.
@codeexample { <i>description</i> }	Delimits a section with the heading “Example:” (followed by some introductory text) to provide an example of code that uses the function; it should be positioned last in the comment block.
@deprecated { <i>description</i> }	Indicates that the current symbol is deprecated. Use this tag to trigger Doxygen to create a summary list of all deprecated symbols automatically. It is important to indicate the version from which the deprecation is effective (for example, use @deprecated since version 2.5.1).
@endcode	Ends a code section begun by @code.
@example { <i>file-name</i> }	Instructs Doxygen to insert a link to this file for an example of code using the API.
@file { <i>description</i> }	Must be defined at the beginning of each source code and header file to describe the content of the file. Description text must start on the next line.

Table F-2 **Alphabetical List of Doxygen Tags (continued)**

Command	Recommended Usage
@fn { <i>function definition</i> }	Allows a comment block to be associated with the given function (global or member of a class) when the comment block cannot be placed just before the actual definition. This is commonly used in Component Framework documentation for APIs implemented via IOS registries, where the actual interface function definitions are generated by the registry compiler. The function definition must exactly match the source code function definition with which it should be associated, including spacing, return type, parameter types, and parameter names.
@history { <i>description</i> }	Provides symbol history information: when it was created, history information on content changes, and whether and when it was deprecated. Example information to be supplied using this tag includes: "New in Release 12.2S" or "Deprecated in 12.4T" and the subsequent descriptive text. The script will treat it as part of regular text block output.
@image html { <i>file</i> }	Include graphics in the generated HTML page.
@invariant { <i>description</i> }	Indicates any invariant condition for the associated symbol. This information can be very useful for unit testing and for proper understanding of the behavior.
@note { <i>description</i> }	Describes a note.
@opaque_struct { <i>description</i> }	Specifies that the structure type being documented is opaque.
@opaque_ptr { <i>description</i> }	Specifies that the pointer type being documented is opaque.
@os_semantics { <i>description</i> }	Describes conditions that are OS-specific.
@param[in] <i>name</i> { <i>description</i> }	Indicates an input parameter. <i>name</i> must be the parameter name, not the type.
@param[in,out] <i>name</i> { <i>description</i> }	Indicates an input/output parameter that is passed as an address because the value it references might be updated within the function.
@param[out] <i>name</i> { <i>description</i> }	Indicates an output parameter that is passed as an address because the value it references might be updated within the function.
@performance { <i>description</i> }	Describes any performance consideration.
@pre { <i>description</i> }	Describes any conditions to be met before invoking a function.
@post { <i>description</i> }	Describes any conditions to be met after the function invocation returns.
@return { <i>description</i> }	Describes the return code of the function. The @retval can also be used when discrete return values need to be documented.

Table F-2 **Alphabetical List of Doxygen Tags (continued)**

Command	Recommended Usage
<code>@retval return-value {description}</code>	Describes a particular return value.
<code>@safety {description}</code>	Descriptive text must begin on the same line as this tag. Describes if the function is interrupt/thread safe or not. Specify one of “#NotThreadSafe”, “#ThreadSafe” or “#Reentrant”. In order to hyperlink to the corresponding topic, define it in xos_defs.h
<code>@see {link}</code>	To refer to another symbol. Specify a single symbol or comma-separated list of symbols for the link.
<code>@semiopaque_struct {description}</code>	Used to specify that the <code>struct</code> type being documented is semi-opaque.
<code>@semiopaque_ptr {description}</code>	Used to specify that the pointer type being documented is semi-opaque.
<code>@since {description}</code>	Describes when (date or version) the symbol was created.
<code>@usage {description}</code>	Subheading for API usage guidelines.
<code>@warning {description}</code>	Inserts a warning when using the symbol.

Note Some of the commands in the above table are not Doxygen native commands, but Doxygen aliases that are defined in the Doxygen template file used for generating Cross-OS API documentation. For example, `@safety` is an alias that is equivalent to `@par Safety:`.

F.8 FAQ

For questions you might have, check the FAQ at:

<http://twiki.cisco.com/Projects/XOS/RunDoxygenFaq>

F.9 How Can I Get Help on Doxygen?

For any questions, contact doxygen-trolls@cisco.com or xos-internal-dev@cisco.com.

Glossary

For information on network software terms from many sources, see the global glossary at:

<http://wwwin-enged.cisco.com/ecm/index.html>

The following terms are used in this book:

A

AAA

Triple A - Authentication, Authorization, and Accounting.

Abstract Syntax Notation 1

See ASN.1.

AC

Attachment Circuit

access control list

(ACL) List of packet filtering rules to provide security features. Standard ACLs only provide filtering based on source IP address adjacency.

access list

List kept by routers to control entry or exit from the router for a number of services, for example to prevent packets with a certain IP address from leaving on a particular interface.

accounting management

Subsystems that are responsible for collecting network data about resource usage. (One of five categories of network management defined by ISO for the management of OSI networks.)

Action Function

A function that is part of an IOS component that is called when a line is successfully parsed.

Active RP

The RP that controls the system, runs the routing protocols, and presents the system management interface.

adjacency

Two nodes are said to be adjacent if they can reach each other via a single hop across a link layer. The adjacency database is a table of adjacent nodes, each entry holding the Layer-2 MAC-rewrite information necessary to reach the adjacent node. The entries in an adjacency database are called *adjacencies*. See also CEF and FIB.

agent

In SNMP, a process on a managed system that answers requests from a manager.

alignment correction

Alignment errors are caused by misaligned reads and writes. For example, a two byte read where the memory address is not an even multiple of two bytes is an alignment error. Alignment errors are caused by a software bug. Alignment errors are reported in the log and recorded by the router. Output from the **show alignment** command provides a record of these errors along with potentially useful tracebacks. The tracebacks for alignment errors can generally be decoded to reveal the function causing the alignment problems. The **show alignment** command is hidden and undocumented. The command is also not supported on all platforms (only high-end routers support it). Alignment errors can generally be corrected by software and, if so, will not cause a crash. Correcting alignment errors does, however, consume processor resources and can result in a performance penalty. If there are continuous alignment errors, the router can spend most of its time fixing them, increasing the CPU utilization. These errors are corrected at interrupt.

ARP

Address Resolution Protocol.

ASICs

Application Specific Integrated Circuits

ASN.1

Formal language used by SNMP.

asynchronous notification

Proactive message from an agent to manager providing information to the manager.

ATM

Asynchronous Transfer Mode.

AToM

Any Transport over MPLS.

atomic lock

A flag that can be set or cleared via an uninterruptable action. See also managed semaphore, semaphore, and watched semaphore.

autonomous switching

Feature on Cisco routers that provides faster packet processing by allowing the ciscoBus to switch packets independently, without interrupting the system processor.

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

backend

Any software performing either the final stage in a process, or a task not apparent to the user.

base class

See class.

base contributor

This is the version of the element that is used as the “common point” for the merge. Typically, this is the version seen in the old REF_LABEL.

base label

The base label is a floating label (i.e. the opposite of a fixed label) that is used to dynamically track the ref label of the child branch. As part of each sync, the base label is moved so that it corresponds to the new ref label. By convention, the base label is the same as the branch name, in ALL CAPS (e.g., the base label for projectx is PROJECTX).

baseline

A release branch (or a stable branch) used as the starting point for new feature development.

binary tree

Data structure suitable for storage and keyed retrieval of information.

bit field

A contiguous array of binary digits (bits) that have individual significance.

bleed-through

A bleed-through file is an element or file that is not branched on the child, that is, has not been modified on the child. For these elements we see the parent’s or ancestor’s version because bleed-through files are not merged during a sync.

Blind Sync

A Blind Sync is a redundant system command synchronization method that sends the same command that was entered on the Active RP to the Standby RP regardless of whether the command execution is successful or not.

blocking

A task blocks when it waits for an event. Except for the implicit blocking that occurs when interrupt service is performed, all blocking in IOS is explicitly requested by the caller.

BM

Boolean or Bit Manager - IOS infrastructure to support dynamically allocated flags (Booleans) for IOS data structures.

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

An area in Cisco IOS memory that holds a packet while it is manipulated by a network device. Internally, the generic pool management code often refers to pool items as “buffers” because that is its primary application: buffer management.

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.

bus error

A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error.

C

CANA

Cisco Assigned Numbers Authority. Group that assigns MIB branch numbers within the Cisco branch to Cisco developers.

card

A module that is inserted into an IOS system that provides network interface support and, in one case, route processing.

CASA

Cisco Appliance Services Architecture. CASA is a protocol designed to allow network appliances to selectively control the flow of IP packets through a router, switch or other network device.

CASE

A CASE service is a runtime replacement for a C switch statement. It reads through a list of functions until it finds the matching service.

Caselist

The run-time replacement for a list of callbacks that are executed sequentially.

Caseloop

The run-time replacement for a list of callbacks that are executed sequentially until one of the callbacks return a TRUE value.

CCITT

Consultative Committee for International Telegraph and Telephone. International organization responsible for the development of communications standards. Now called the ITU-T. See also ITU-T.

CCN

CCN (Configuration Change Notification) includes the config logger that is used to track and report configuration changes.

CEF

Cisco Express Forwarding. Switching mechanism with performance on a par with fast-switching but that also scales to support internet backbone requirements. CEF uses a forwarding information base (FIB) and an adjacency table. *See also* FIB, adjacency, and dCEF.

CF

Checkpoint Facility.

Checkpointing

Refers to the saving (that is, synchronization) of client-specific state data which will be transferred to a peer client on a remote RP. Once a valid Active to Standby peer client checkpointing session is established, the checkpointed state data will be guaranteed to be delivered to the remote peer client at most once, in order, and without corruption.

child contributor

This is the version of the file on the child that contributes to create a merged file. **findmerge** checks out this version of the file to place the merged file.

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.

Cisco 1000-12000 router

(1000-12000 line) The x-thousand product line consists of hardware that is tailored to specific markets: the 12000 is for optical backbones and high performance; the 7000 is for enterprise backbones or the Internet edge; the 4000 is for concentrators and filtering on the edge of the corporate backbone; the 3000 and 2000 are for regional and remote sales offices; and the 1000 is for telecommuters and single-person offices.

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.

clone

Cloning is the process that is used to exactly replicate a branch. The cloned branch will have the same parent (REF_LABEL) and contents as the original branch. This is only done in some special cases.

cold reboot

An IOS reload wherein the ROM Monitor (ROMMON) copies the configured image from a storage device (like flash memory) into main memory, the image is decompressed, and execution is started.

collapse

Collapsing is the process of delivering all the changes on a child branch (feature additions and fixes) back to its parent. (Right before the collapse, a terminal sync is performed and the latest code on the parent is integrated into the child.)

config spec

The file used to specify the configuration used by a particular view.

configuration

A specific set of versions of source files. Normally, a configuration is somewhat permanent in that there is a desire to be able to recreate the configuration at any time in the future. A good example of the use of configurations is the weekly builds of the main release trains performed by the build group.

contiguous DMA (packet buffers)

The basic packet manipulation method in which an entire packet is stored in a single buffer. This method is less memory efficient than scatter-gather DMA (packet buffers).

contributors

These are the three versions of an element that together create a merged file. During a sync, all changes made between the base contributor and parent contributor are incorporated into the child contributor to create the merged file. (The base contributor can be a blank file, but then, that is the contributor.)

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 non-modular 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.

CPUHOG

CPU hog occurs when the process runs for too long. If a process runs for more than two seconds, a CPUHOG message is printed to the console. Users can modify this value of two seconds using a hidden global config command, **scheduler max-task-time ms**; where 'ms' is the maximum time processes allowed to run without suspending before a CPUHOG error is reported for the processes.

cruff

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

dCEF

Distributed CEF. To improve the scalability of high-end routers, the CEF tables are distributed to special intelligent line cards, such as VIP line cards (7500) or Gigabit Switch Router (GSR) line cards. *See also* CEF, FIB, and adjacency.

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.

DEN

Directory Enabled Networking. DEN specification describes an information model of network elements, protocols, services, and their relationships. A key foundation for policy-based networking and intelligent network services, DEN can fundamentally change how interoperable network applications are developed and used.

direct queue

Singly linked list in which the first longword of the data structure is reserved for linking together the items in the list.

distributed environment

Environment in which slave processors perform interrupt-level route processing under the direction of a central route processor (RP). The slave processors are on line cards (*see*). Examples of distributed environments: the 7500, 12000, and 10000 series. Compare with non-distributed environment.

DLCI

Data Link Connection Identifier.

DMA

Direct memory access. The transfer of data from a peripheral device, such as a network interface card, into memory without that data passing through the microprocessor. DMA transfers data into memory at high speeds with no processor overhead. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.)

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.

DPM

Defects Per Million.

Dual-RP

Dual Route Processors are a requirement for the one-to-one redundancy used to achieve HA.

dynamic pool

A pool for which the number of items it contains can change over time to reflect the current needs. 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.

E

element

An element is an object in ClearCase which encompasses a set of versions organized into a version tree. In ClearCase, an element includes both files and directories.

ELMI

Enhanced LMI.

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.

F

FASTCASE

A FASTCASE service is basically a function vector table. There is no boundary checking performed on the indices passed.

fast switching

Cisco feature in which a route cache is used to expedite packet switching through a router. If fast switching is enabled, the driver code will transfer control temporarily to the fast switching code, which searches the route cache for a frame and other information constructed from a previously transmitted packet. If the route cache contains an entry, the fast switching code will attempt to send the packet directly to the destination interface. If the interface is busy, the packet is placed on the queue for that interface. There are several types of platform-specific fast switching techniques. *Compare with process switching.*

FIB

Forwarding information base (common ISO usage). Database of information used to make forwarding decisions. It is conceptually similar to a routing table or route-cache but very different in implementation. *See also CEF and adjacency.*

FP

Forwarding Processor.

FT

Function Table, a structure containing function pointers and other constant data, defining standard methods to be accessed from base class code. Each subblock has a pointer to a FT, defining the subblock type and pointers to the methods of the subblock class.

frame header

One of the headers in a data buffer, such as those specified in IEEE 802.3 and 802.5.

FSMs

Finite State Machines

function table

(ft) Static data structure that describes how a feature uses the FIB subblock facility.

function vector (FV)

A pointer to a routine that performs a unique function. Usually FVs are associated with a network interface and many are stored within the hardware IDB (hwidb). Some examples are the routines to perform the network-interface-unique commands **show interface** and **show supports**; the code to enable and disable a network interface; and the media-specific routines to perform encapsulation (build a new frame header) and decapsulation (interpret the network-layer protocol of a packet).

G

GAS

GNU Assembler, supported by Cygnus.

GCC

GNU compiler, supported by Cygnus.

GDB

GNU debugger, supported by Cygnus.

GSR

Gigabit Switch Router

H

HA

High Availability.

HA-aware

A feature or protocol is said to be “HA aware” if it has been designed or modified to support, either completely or partially, undisturbed function through an RP switchover.

HA-unaware

A feature or protocol is said to be “HA unaware” if it has not been designed or modified for SSO.

header pool (packet buffers)

A pool of Cisco IOS packet header items only, which are defined by the `paktype` structure. There are no packet data buffers associated directly with each packet header item in the pool.

heap

Memory that remains in a region after an imaged has been loaded.

Hitless Software Upgrade

See HSU.

Hot ICE

Hot ICE (Hot IOS Configuration Enhancements) is the first stage to provide a more robust and functional means of automated configuration and provisioning for IOS. Hot ICE includes support for Access Session Locking, BEEP/TLS, and HTTP/S.

HSU

The goal of Hitless Software Upgrade is to enable upgrading or downgrading the running IOS code image on a router with no loss of the sessions. It is targeted at reducing the DPM associated with planned upgrades. Full Hitless Software Upgrade of IOS on the RP requires dual RPs in order to operate, whereas Hitless Software Upgrade of Line Card software can only be achieved with redundant line cards.

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.

IF-MIB

Interfaces Group MIB (Management Information Base). The current specification for the IF-MIB is found in RFC 2233. The MIB module to describe generic objects for network interface sublayers. This MIB is an updated version of the MIB-II if Table, and incorporates the extensions defined in RFC 1229.

IFS

IOS File System.

ILIST

An intermittent list service is similar to a LIST service except that a special callback can be specified at invocation which is called between callbacks in the list. This ability is more commonly used to allow the invoking task to suspend on large lists that could otherwise become a CPU hog.

ILMI

Integrated Local Management Interface.

in-band signaling

Transmission within a frequency range normally used for information transmission. Contrast with OOB (out-of-band signaling).

incremental sync

A sync performed as a stand-alone process to update the child with the parent's changes. The changes can include bug fixes and/or new features.

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 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).

input queue of an interface

Cisco IOS software refers to the count of the number of packets in the system that are associated with a particular interface as that interface's *input queue*, although this structure is not actually represented internally as a queue. Packets that either came into the system from an interface or had ownership transferred to an interface are considered to be *charged* to that interface and are credited to the "input queue" count, which is maintained in the `hwidb` for the interface.

instance identifier

Designation of a specific occurrence of an object in the MIB tree. Also called an object identifier.

integration branch

The ClearCase branch to which changes are committed. This type of branch is used for shared development by a team working closely on a project or a release. Integration branches are permanently stored in ClearCase.

interface or network interface

The hardware or firmware that provides a mechanism to move packets between the network link and storage (memory) on the network device.

interface driver or network interface driver

The software component of a Cisco IOS image that controls the interface.

Interface Processor

(IP) Before VIPs came into being, all the different 7000 and 7500 cards were known as "IPs." In fact, VIP stands for "Versatile Interface Processor." For example, the ATM Interface Processor (AIP) is the ATM network interface for the 7000 series, designed to minimize performance bottlenecks at the UNI (User-Network Interface, an ATM Forum term that goes with NNI, Network-to-Network interface, and SNI, Subscriber-Network Interface). Other IPs include the Channel Interface Processor (CIP), Ethernet Interface Processor (EIP), Fast Ethernet Interface Processor (FEIP), FDDI Interface Processor (FIP), Fast Serial Interface Processor (FSIP), HSSI Interface Processor (HIP), MultiChannel Interface Processor (MIP), Serial Interface Processor (SIP), and Token Ring Interface Processor (TRIP).

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.

IS-IS

Intermediate System-to-Intermediate System interior gateway routing protocol.

ISSU

In Service Software Upgrade.

ISV

An ISV (independent software vendor) makes and sells software products that run on one or more computer hardware or operating system platforms.

ITU-T

International Telecommunication Union Telecommunication Standardization Sector.

International body that develops worldwide standards for telecommunications technologies. The ITU-T carries out the functions of the former CCITT. See also CCITT.

J

jitter

Method of randomizing an expiration time within set limits.

jump table

A function table that contains a set of functions that all accept the same parameters and return the same value.

L

L2F

Layer Two Forwarding Protocol.

L2TP

Layer 2 Tunneling Protocol.

label

A label is a mechanism provided by ClearCase to assist in capturing configurations. A label is added to the individual versions of a configuration.

LDAP

Lightweight Directory Access Protocol. Protocol that provides access for management and browser applications that provide read/write interactive access to the X.500 Directory.

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.

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.

line card

(LC) "LC" implies some autonomous intelligence and independent processing, as with the VIP cards for the 7500 series and, later, all of the GSR line cards. When an LC has a CPU onboard, it does route processing with the assistance of CEF chip-sets built into the board. LCs are wide cards, able to handle many interfaces and having multiple physical interface types.

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

A LIST service is the runtime replacement for a list of C functions that are executed sequentially. It reads through a list of functions, calling one function at a time.

list manager

Set of functions for manipulating doubly linked lists.

LMI

Local Management Interface.

load balancing

In routing, the ability of a router to distribute traffic over all its network ports that are the same distance from the destination address. Good load-balancing algorithms use both line speed and reliability information. Load balancing increases the utilization of network segments, thus increasing effective network bandwidth.

LOOP

A LOOP service is a runtime replacement for a C while loop. Each function registered for the LOOP service is called until one of the functions returns TRUE.

M

Maintenance Mode

A system state in which both RPs are present but logically separate and not running the HA algorithms. The operator may place the system in split mode for maintenance or software upgrade (that is, HSU).

managed

An adjective used by IOS Event Management Services to identify a category of object that participates in Event Management. See also managed Boolean, managed queue, managed semaphore, and managed timer. Synonymous with watched.

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 an atomic lock 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, atomic lock.

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.

MCEF

Multi CEF (Cisco Express Forwarding). CEF running with multiple forwarding FIBs (Forwarding Information Base) as part of VPN configurations.

memory management unit

See MMU.

memory pools

Pools used to manage heaps.

merge

This is the generic term for the act of combining different strands of development, with the specific meaning being determined by the context. In IOS development, a combination of Clearool **findmerge** and RCS **merge** is used to merge code. These tools work on the affected elements one at a time to complete the merge. Merging plays a key role in all branch integration processes.

message

In the scheduler, a simple interprocess communication (IPC) mechanism that 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.

MFI

MPLS Forwarding Infrastructure.

MIB

Management Information Base. Abstract database description that defines all the information about a managed system that a manager can view or modify.

MMU

A Memory Management Unit (MMU) is a hardware device used to support virtual memory and paging by translating virtual addresses into physical addresses.

MPLS

Multi-Protocol Label Switching.

MTBF

Mean-Time-Between-Failures is the average expected time between failures of a product, assuming the product goes through repeated periods of failure and repair.

MTTR

Mean-Time-To-Repair is the average expected time to restore a product from a failure.

MTU

Maximum Transmission Unit. Maximum size of any frame that can be transmitted on a particular media.

multicast ports

In IP Cs, an aggregation of ports referenced as a single port so that messages can be transmitted from one source to multiple destinations.

N

NDB

Network descriptor block. Mechanism used by a routing information base (RIB) to distribute the best route to a prefix to all of its clients. (From the RIB perspective, CEF is a redistribution client.) *See also RDB.*

network controller

1. CPU on board the network interface module that carries out status and control functions and performs intelligent operations 2. Component of the network interface that communicates with the core CPU.

Network descriptor block

See NDB.

network header

In a data buffer, the protocol datagram header; for example, IP, AppleTalk, or IPX headers. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code.*)

network interface

1. Boundary between a carrier network and a privately-owned installation 2. Unit (card) that plugs into a router slot and is used for the sending and receiving of packetized data. Also called a network interface module (NIM.) 3. Network protocol, for example Ethernet, Fast Ethernet, Token Ring, Serial, ISDN-BRI, ISDN-PRI, T1, E1, HSSI, IBM channel, ATM, FDDI
4. Hardware component of the unit that implements the physical layer. Also called the *port*.

Network Interface Module (NIM)

See NIM.

Network Time Protocol (NTP)

See NTP.

NIM

A non-hot-swappable card that is installed in the c4000 and c4500 series. It is relatively “low tech” in that it can handle only one physical layer type per NIM, for example four Ethernet ports and two Token Ring. NIMs sometimes have hardware assist but it is limited to physical layer framing. The NIM may be considered a forerunner to the PA, except that it is only usable on the c4000/c4500 series and the NIMs themselves do not usually contain any intelligence.

While the term “NIM” is not used to describe any of the cards inserted into the 7000-7500 series, a smaller entity used by the older 7500 IPs had a similar name, “network module.” These were for configuring different interface types, for example the FSIP had one that supported four ports, so that you could have a 4-port FSIP, or an 8-port FSIP if you installed two. These are not to be confused with “network interface modules (NIMs).”

NM

Network Management.

NMI

Non Maskable Interrupt.

NMS

Network Management Station.

nondistributed environment

Environment in which a single route processor (RP) performs all routing tasks (without the help of slave processors.) *Compare with distributed environment.*

NRP

Node Route Processor.

NSF

Non-Stop Forwarding.

NSP

Node Switch Processor.

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).

NVRAM

Non-Volatile Random Access Memory.

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.

OCE

Output Chain Element.

OID

Object identifier.

OIR

Online Insertion/Removal, hot swap.

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

packet

Data that is either to be transported over a network (forwarded or switched), originates in the local network device (such as ARP messages exchanged with neighboring devices), or is “consumed” on the local network device (such as SNMP control messages). Packet and frame are sometimes interchangeably used, although in this document, frame usually refers to DLL data encapsulated in a packet.

packet buffer

A Cisco IOS method to represent and manipulate packets using two elements, a structure to hold the packet data and a `paktype` header structure that maintains significant information about and references into the packet data.

packet buffer pool

A pool of packet buffers. “NAME packet buffer pool” refers to one of the 6 different size pools named for their item size that are created during system initialization for use by any system component. These packet buffers are used for jobs such as console logging, manipulating packets that originate on the local network device (for example, ARP inquiries), and fallback packet buffer allocation when an interface’s private pool has no available buffers.

Packet Buffer Services

The Cisco IOS code that manages packet and particle-based buffers using the Cisco IOS pool management code.

page fault

[virtual memory] an access to a page (block) of memory that is not currently mapped to physical memory.

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. See also page fault, MMU, virtual memory.

PAM MBOX

Port Adapter Module Mail Box. The shared memory space between NSP and NRP that is used as a communication path.

PAM MBOX interface

The pmbox IDB interface on top of PAM MBOX capable of forwarding IP traffic.

parent contributor

This is the version of the element on the parent branch that contributes to the merge. This is the version of the element seen in the new REF_LABEL (attribute on the child branch).

parent timer

In managed 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.

particle

A Cisco IOS storage element that holds a portion of a packet implemented by a scatter-gather DMA (packet buffers) method. A particle consists of a particle header structure, partictype, and a particle data buffer, and represents part of a contiguous packet. Particles are connected to a packet buffer header in a chain of evenly-sized blocks that are smaller than the maximum MTU-size packet of an interface, and represent the entire packet with "scattered" blocks located randomly in memory.

particle-based packet

A particle chain that represents an equivalent, contiguous packet, attached to a packet buffer header.

particle chain

A list of particles linked together by fields in the partictype header that represents part or all of a particle-based packet.

particle pool

A pool that contains particles and implements scatter-gather DMA (packet buffers).

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.

patch

This is a process in which the file and directory diffs that are present in a "patch file" are applied to a branch. The patch file is typically produced by the cc_diff tool and the patch is done by the cc_patch tool.

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.

PID

Process identifier.

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.

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).

Platform-dependent Code

Platform-dependent code is compiled for a specific platform. Platform-dependent code contains instructions, idioms, etc., that are specific for a given platform, and thus, the compiled binary works only for that platform. For example, platform-specific code and platform-generic code is platform-dependent code.

Platform-generic Code

Platform-generic code uses semantics that are specific to the platform. However, it is *not* obvious upon inspection that such code is designed to execute on a specific platform. Platform-generic code depends exclusively upon conditional compilation paths to produce binaries that are specific to a given platform. As such, platform-generic code is a type of "template" where algorithms remain the same but the data manipulated differs. These platform-generic "templates" should be considered as part of the platform-generic API shared by all platforms. For example, platform-generic code is used for the same purpose by multiple platforms, but is customized per platform by separate platform-specific `cisco_<platform>.h` (see the `machine` directory) header files and/or by conditional compilation, and the resulting object code is platform-specific. For example, `interface.c` is platform-generic code. Also, the `ipfast_flow_les.c`, `ipfast_frag.c`, and `isdn.c` files are platform-generic.

Platform-independent Code

Typically, platform-independent code is independent of both the CPU and RTE (run-time environment). However, the generated binaries are *always* CPU-specific. Examples of platform-independent code include routing protocols, the scheduler, the error logger, and other high-level features. For instance, the `main()` function is platform-independent.

Platform-specific Code

Platform-specific code uses semantics that are specific to the platform. It is obvious upon inspection that platform-specific code is designed for a given platform. Typically, files and/or directories have the platform name in the file name. Examples of platform-specific code are the bootstrap code, device drivers, and Flash memory drivers. Also, typically included in platform-specific 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. The initialization of various hardware devices, such as timers, interrupt controllers, and port adapters, is also performed by platform-specific code. For example, the `platform_memory_init()` function is platform-specific.

plug-in

Thankfully, within the IOS software world, we have a generic term. A “plug-in” is anything that plugs in, regardless of whether it's a line card, port adaptor, port module, or other type of card.

pool

A container in which to hold and manage similar items. The container structure holds items that are available or not currently in use; items that are in use maintain a reference back to the container so they can be returned to the pool when they are no longer in use. Data structures such as buffers are managed in pools. The Packet Buffer and Particle Services are two main IOS buffer pool clients, and maintain packet buffers or particles in pools.

pool cache

An array of pointers to pool items that allows faster retrieval of available items. For buffer pools, they are generally used only with private pools by interface driver code.

pool client

Code that uses the Cisco IOS pool management code directly and calls pool API functions to manage items, for example, to provide a pool-based service such as the Packet Buffer or Particle Services. Also can refer to a second level of pool clients: clients who use a pool-based service (a pool client service) by calling pool wrapper functions, and thus are indirectly clients of generic pool management functions.

pool client services

Pool clients such as the Packet Buffer or Particle Services, that provide a pool-based service through wrappers around generic pool API functions.

port

In IPCs, a communications end point.

port

A port incorporates changes made in a source branch or a label into a target branch. Depending on the situation, either all changes or a subset of that is incorporated into the target. In all ports, the entire set of source and target branches are unrelated; that is, one is not a direct descendant of the other. The changes from the source or label are either merged into the target (using Cleartool `findmerge` and RCS `merge`) or patched (using `cc_diff` and `cc_patch`).

Port Adaptor

(PA) Unlike the GSR LCs, VIP LCs have no interfaces integrated into them, so PAs need to be added. PAs are widely used interface cards with a PCI bus connector. Two can be installed per VIP card and the same ones fit into the 7200 platform. PAs can be found in several other places besides the 7200 and VIP, such as the Cat6K FlexWan line card. Although some PAs have intelligence, none do distributed switching like the VIP LC. On the 7200 platform, PAs can be hot-swapped, but not on the VIP. The VIP itself is hot-swappable.

port identifier

In IPCs, a 32-bit integer that uniquely references a communications end point.

Port Module

(PM) The PM is basically a repackaged PA that is used in the 3600 and 2600 series. Electrically, it is exactly the same as a PA. However, for marketing reasons, the metalwork is different: you cannot take a PM and put it in a PA slot. The other difference is that PMs are not designed to be hot-swappable.

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.

POS

Packet Over SONET.

PPP

Point to Point Protocol.

PPPoA

PPP over ATM.

PPPoE

PPP over Ethernet.

PPTP

Point-to-Point Tunneling Protocol.

PRC

Parser Return Code - returned by an action function when done processing a command.

preintegration branch

A preintegration branch is a branch used to integrate a few features before collapsing it to a release branch. The reasons we do it are that we get to test the features in a “safe” environment before code is put on a release branch, and that the release branches are kept relatively clean as only tested code gets committed.

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.

Primary RP

Deprecated. Replaced by “Active.”

priority

Order in which the scheduler executes processes. Processes can run at one of the following priority levels: critical, high, medium, or low.

private pool

A pool of items to which access is restricted. Private pools are generally used to provide better resource control but require more management overhead.

process

A collection of programs and/or data often arrayed under one or more "threads" or "tasks" that can be independently scheduled by the operating system. Historically, "process" was a synonym for task or thread, but this usage has been incorrect since the advent of virtual memory. In systems with virtual memory, a process occupies its own private address space. There is no virtual memory in Cisco IOS (each scheduled task shares memory with all the other scheduled tasks in IOS).

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.

process watchdog

Process watchdog timeouts are used to prevent a process from locking the CPU forever. This timeout is implemented by a scheduler and by a timer interrupt. The scheduler timestamps the process when it is scheduled and the timer interrupt fires every four milliseconds. The duration of timer interrupt firing and the routine that services the timer interrupt is platform dependant.

public pool

A pool of items available to all Cisco IOS components in the system. Public pools provide memory efficiency at the cost of potential resource contention when multiple components share access to pool items. For Packet Buffer Services, for example, typical Cisco IOS platforms allocate several public pools of packet buffers of different sizes at initialization time.

punting

(Ciscoism) Action by a device driver of sending a packet "up" to the next slowest switching level when the attempt at a lower level has not yielded a path or the packet could not be switched at that level for another reason. In the case of CEF, the levels are as follows, from lowest to highest: distributed CEF (dCEF), CEF, fast switching, route processing. For more information on the role of drivers, see "Cisco IOS Architecture" in *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.

PVC

Permanent Virtual Circuit.

PW

pseudowire

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

A Cisco IOS radix tree is a search tree structure that has a search time of O(n), and that decides routing by matching the network address and the network address mask. It is currently used in BGP, PIM and DVMRP tables.

RB tree

Red-Black tree. The Cisco IOS implementation is a threaded tree.

RDB

Route descriptor block. Used by the routing information base (RIB) to pass path information about a given prefix. RDBs are components of the network descriptor block (NDB, see). The NDB describes the route whereas the RDB describes the paths available to the route.

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.

receive ring

Used by hardware devices to buffer packets received on an interface. Each ring entry has a descriptor that contains pointers to the packet buffer, and the packet size and status of the entry, including ownership and error information. Receive descriptors are thought of as connected in a ring. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.) See also transmit ring.

reference label (REF_LABEL)

The label, (for example, a REF_LABEL for geo_t is v123_3_2U) on the parent branch to which the child branch is currently synchronized. A REF_LABEL attribute is attached to the child branch for tracking this point.

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 of 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

A collection of service points to install (register) and execute (invoke) callback functions and discrete values of PIDs. Each registry is frequently placed in a unique subsystem of type SUBSYS_CLASS_REGISTRY.

registry compiler

Preprocessor to generate function prototypes and definitions for all service points in all registries.

registry name

Defined in the BEGIN REGISTRY *REGISTRY_NAME* in the metalanguage. It is referenced when the registry is initialized in the `create_registry_name()` function.

Remote Procedure Call

See RPC.

reparent

Reparenting is the process that changes the parent, and hence the ancestor history, of a branch. A branch is reparented in two steps. First the parent attribute or the child is changed to reflect the new parent. Then a reparent sync is done to a label on the new parent. The label on the new parent should include all the changes made in the old parent. The main requirement for a legal reparent is that the *new parent* should contain all the changes made in the *old parent*. Most often, a branch is reparented to its grandparent (after the collapse of the current parent), or to a sibling (to merge in the sibling's changes without waiting for a collapse).

reparent sync

A sync performed as part of the reparent process after the parent attribute of the branch has been changed. In this type of sync, the child branch, previously synchronized to a label on the old parent, is synchronized to a label on the new parent. The reparent sync completes the reparent process and ensures that the child is able to get code from the new parent.

Reset Hyperlink Procedure

This procedure is run on a branch whose parent has been collapsed. This physically marks in the ClearCase database the logical state of the branches so that findmerge will not perform any spurious merges.

RETVAL

A RETVAL service is identical to the CASE service except that it returns a value instead of a void.

RF

Redundancy Facility.

Route Processor

See RP.

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.

RPC

Remote Procedure Call. Procedure call to an application in which the actual work happens on another processor.

RPR

Route Processor Redundancy - the feature formerly known as EHSA.

RPR+

Route Processor Redundancy Plus - An enhancement to RPR / EHSA in which the standby processor is fully initialized. An RPR+ switchover does not involve linecard reset nor linecard software reload.

RSP

Route/Switch Processor. Processor module that integrates the functions of the RP and SP.

S

SAL

Segment-switching Abstraction Layer.

scatter-gather DMA (packet buffers)

A packet manipulation feature provided by the network interface and Cisco IOS driver software that allows a packet to be read from or stored in multiple buffers, and reconstructed as a contiguous packet for transmission. This feature improves efficiency of memory usage for packet manipulation.

scheduler watchdog

Scheduler watchdog implements the process timeout schedule by timestamping the process with details on schedule and timer interrupts. A maximum time limit of 2 minutes is provided by default for the scheduler to reschedule another process.

The amount of time the scheduler is allowed to run is controlled by a global IOS defined value of 2 minutes. This can be changed by the user by using a global configuration service internal command, **scheduler max-sched-time ms**.

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.

Secondary RP

Deprecated. Replaced by “Standby”.

Segment Switching Manager

SSM is a data-plane process that was plugged into the L2HW (Layer 2 Hardware) Switching API between the L2 Hardware Switching API and the hardware.

semaphore

Memory location that is used by multiple processes to serialize their access to one or more resources. The resource can be anything, for example Flash memory or the table of IP routes. See also managed semaphore, atomic lock, watched semaphore.

service

A type of service point, for example, CASE, LIST, or LOOP. However, in common usage, the term "service" really means "service point".

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.

service point name

Defined in the `DEFINE my_svc_point_name` statement in the metalanguage. It is expanded to define the registry functions required for this service point, such as

`reg_add_my_svc_point_name()` and `reg_invoke_my_svc_point_name()`.

signal

See exception.

silicon switching

Switching based on the silicon switching engine (SSE), which allows the processing of packets independent of the silicon switch processor (SSP) system processor (*see*). Silicon switching provides high-speed, dedicated packet switching.

silicon switching engine

(SSE) Routing and switching mechanism that compares the data link or network layer header of an incoming packet to a silicon-switching cache, determines the appropriate action (routing or bridging), and forwards the packet to the proper interface. The SSE is directly encoded in the hardware of the SSP (silicon switch processor) of a Cisco 7000 series router. It can therefore perform switching independently of the system processor, making the execution of routing decisions much quicker than if they were encoded in software. *See also* silicon switching and silicon switch processor.

silicon switch processor

(SSP) High performance silicon switch for Cisco 7000 series routers that provides distributed processing and control for interface processors. The SSP leverages the high-speed switching and routing capabilities of the silicon switching engine (SSE) to dramatically increase aggregate router performance, minimizing performance bottlenecks at the interface points between the router and a high-speed backbone. *See also* silicon switching and silicon switching engine.

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.

Simplex

A system state in which only a single RP of a redundant pair is operational. Normally, this RP will operate as the Active RP.

singly linked list

See queue.

SMI

Structure of Management Information. Defines the components of a MIB and the formal language for describing them.

SMP

Symmetric Multi-Processor. Where more than one CPU core shares a single system image and where any CPU can perform any operation in the system.

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.

software watchdog

A software watchdog timeout happens when the user disables all interrupts (including the NMI), and does not interfere with the hardware watchdog register. Note that the `raise_interrupt_level(ALL_DISABLE)` is different from `disable_interrupts()`.

SP

Switch Processor. Cisco 7000-series processor module that acts as the administrator for all CxBus activities. Sometimes called *ciscoBus controller*. See also CxBus.

Split

See Maintenance Mode.

spurious accesses

Spurious access is an attempt by Cisco IOS software to access memory in a restricted location. An example of system log output for a spurious access is shown below:

```
%ALIGN-3-SPURIOUS: Spurious memory access made at 0x60968C44 reading 0x0
%ALIGN-3-TRACE: -Traceback= 60968C44 60269808 602389D8 00000000 00000000
00000000
00000000 00000000
```

A spurious access occurs when a process attempts to read from the lowest 16 KB region of memory. This portion of memory is reserved and should never be accessed. A read operation to this region of memory is usually caused when a nonexistent value is returned to a function in the software, or in other words, when a null pointer is passed to a function.

SSG

Service Selection Gateway.

SSM

Segment Switching Manager.

SSO

Stateful Switchover.

SSS

Subscriber Service Switch.

stable label

A label on a branch that indicates that the branch content up to that point has undergone appropriate testing and validation, as determined by the branch manager. These labels are usually the recommended sync points for child branches. The criteria for assigning a stable label depends on the person or team managing the branch.

Standby RP

The RP that waits for the active or primary RP to fail.

State

If an action function has multiple exit points that are considered successful, each of these are considered to be a separate state.

static pool

A pool for which the number of items it contains remains the same.

Structure of Management Information

See SMI.

STUB

A STUB service takes zero or one functions (like a LIST service) and can return a value (like a RETVAL service).

STUB_CHK

A stub check (STUB_CHK) service is similar to a STUB service except that it protects against the unintentional overwriting of an existing stub.

subblock

See IDB subblock.

SUBSYS_CLASS_REGISTRY

Class of the subsystem that is initialized during network device startup and is used to create each registry using the `create_registry_registry_name()` function. For more on subsystem classes, see Chapter 13, “Subsystems.”

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.

SVC

Switched Virtual Circuit.

Switchover

An event in which system control and routing protocol execution is transferred from a failed processor to a standby processor.

Switch Processor

See SP.

sync

Sync is the process of incorporating the changes that have been made in the parent branch as of a particular label into a child integration branch. Afterwards, the child is said to be “sync’d” to the parent as of that label, and that label is the child’s new REF_LABEL (attribute).

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

task

A unit of work recognized by the IOS scheduler as potentially needing system resources.

task branch

A ClearCase branch that is owned and controlled by an individual user. This type of branch is short-lived and is removed once the changes have been committed into an integration branch.

terminal sync

A sync performed as part of the collapse process. This sync is typically done to the latest available label on the parent. This is the last sync performed on a branch before it is collapsed into the parent.

text segment

The text segment contains all the executable code (assembler instructions) contained in an image.

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 multi-character operator.

trap

Type of asynchronous notification in which unacknowledged datagrams are sent by the agent to the manager.

transmit ring

Used by hardware devices to buffer packets to be transmitted over an interface. Each ring entry has a descriptor that contains pointers to the packet buffer, packet size, and status of the entry, including ownership and error information. Transmit descriptors are thought of as connected in a ring. (For more information, see *Cisco IOS Network Interface Drivers: Fundamentals of Architecture and Code*.) See also receive ring.

U

Unicast RPF

(unicast Reverse Path Forwarding) The goal is to verify if the source IP is reachable for the purpose of preventing malformed or forged source IP addresses from entering a network.

USM

User-based Security Model.

UTC

Coordinated Universal Time, also known as zulu time, formerly Greenwich Mean Time (GMT). Time zone at zero degrees longitude.

V**VALUE**

A VALUE service is a lookup table of 32-bit values.

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 behavior of the operating system.

VFI

Virtual Forwarding Instance.

VIP

1. Versatile Interface Processor. Interface card used in 7000 and 7500 series routers. Provides multilayer switching and runs the Cisco IOS software. 2. virtual IP. Enables the creation of logically separated switched IP workgroups across the switch ports of a Catalyst 5000 running Virtual Networking Services software (on some Catalyst 5000 switches, enables multiple workgroups to be defined across switches and offers traffic segmentation and access control).

Virtual Ifc

Virtual Interface, for example, a virtual-access or tunneling interface.

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).

VPLS

Virtual Private LAN Service

VRF

VPN Routing/Forwarding. The mechanism to configure multiple routing tables and MCEF functionality.

VRFifying

Slang used to describe the software task of enhancing a protocol or application to work with MCEF technology.

W

WAN Interface Connector

See WIC.

warm reboot

An IOS reload without ROM Monitor (ROMMON) intervention, wherein the image restores the read-write data from a previously saved copy in the RAM, and starts execution. This does not involve a flash-to-RAM copy and image self-decompression.

warm reboot storage

The RAM area used to store the data segment and bookkeeping information required for the warm reboot feature to work.

watched

See managed.

watched Boolean

See managed Boolean.

watched queue

See managed queue.

watched semaphore

See managed semaphore. See also semaphore, atomic lock.

WIC

A daughter card arrangement on the 3600 and 2600 series that allows a better mix of interfaces for slower speed WAN interfaces such as ISDN.

working set

[memory management] The set of all pages used by a process during some time interval.

X

X.500

ITU-T recommendation specifying a standard for distributed maintenance of files and directories.

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.

P A R T 8

Index

Index

Symbols

! symbol 1-cxii
 #define's, setting up #include's and 19-7
 #include Directives A-22
 %p, format specifier for a pointer A-30
 **previous 4-47
 ^ character 1-cxii
 ctype array 11-17

Numerics

10000
 ESR 23-27
 7000-specific systems
 Autonomous and Silicon Switching 23-12
 overview 23-12
 Route Processor 23-13
 Switch Processor 23-13
 7200
 7200L720x cards 23-24
 Parallel eXpress Forwarding (PXF) 23-24
 VXR/NSE-1 cards 23-24
 75xx, 720x/71xx systems 23-13

A

AAA
 accounting call flow 31-12
 accounting functions 31-10
 accounting model using RADIUS
 (figure) 31-16
 architecture
 (figure) 31-14
 architecture diagram 31-13
 attribute functions 31-1
 authentication and authorization functions 31-6
 authentication call flow 31-10
 authorization call flow 31-12

authorization method 31-12
 basic configurations 31-16
 functions 31-1
 history 23-1
 PPP configurations 31-17
 process interaction
 (figure) 31-15
 process interaction diagram 31-14
 profile functions 31-7
 server group functions 31-8
 using RADIUS 31-15
 utility functions 31-3
 Abstract Syntax Notation 1
 See ASN.1
 Abuse of the Cisco IOS Infrastructure A-37
 AC 23-81
 access lists
 in distributed environment 23-34
 NetFlow Switching 23-18
 overhead of 23-34
 processing 23-34
 accounting
 NetFlow switching 23-18
 See AAA
 ACLs
 See access lists 23-18
 activatehigh flag D-4
 active RP
 defined 24-2
 add_default_alias function 27-79
 address filter function vectors 6-19
 address intervals
 virtual memory 4-76
 address spoofing
 preventing (RPF) 23-34
 addresses
 AppleTalk 17-8
 Banyan VINES 17-10
 IPv6 17-12
 adjacency (CEF)
 special types 23-47
 table, populating 23-20

adjacency tables
 CEF 23-20
algorithms, designing A-36
aliases
 email archives, searching 1-18
alignment, data
 checking A-30
 portability issues 26-6
alternate mode
 parser 27-75
ambiguity
 parser, debugging 27-80
ANSI
 string functions 17-1
ANSI C A-17
ANSI C library functions 1-ix, 11-16
ANSI C, using A-17
AppleTalk
 addresses 17-8
Application Specific Integrated Circuits
 See ASICs
archives
 email aliases, searching 1-18
Arithmetic Overflow A-30
arithmetic overflow, checking for A-30
arithmetic, pointer, performing A-30
arrays
 secure mode
 comparing 11-10
 copying 11-10
 copying to strings 11-10
ASICs 23-83
ASN.1, definition 28-2
assembler, inline 26-12
asynchronous notifications, SNMP
 controlling 28-36
 defining 28-35
 description 28-3
 generating 28-38
 informs, definition 28-3
 location 28-35
 snmp-server enable command 28-36
 traps, definition 28-3
Attachment Circuit 23-81
authentication
 See AAA
authorization
 See AAA
auto storage class, using A-27
automore
 changing automore's header in midstream 17-3
 conditionally asking for permission to do more
 output 17-4
 disabling "MORE---" processing 17-3
 finding if user has quit automore 17-3
 turning on automatic "MORE---" processing 17-2
 user response processing 17-2
automore_conditional function 17-4
automore_disable function 17-3
automore_enable function 17-2
automore_header function 17-3
automore_quit function 17-3
Autonomous Switching
 description 23-12
AutoSecure management 30-17
AutoSecure subsystem 30-18
availability
 defined 24-2
AVL trees
 deleting 21-7
 overview 21-2, 21-5
raw
 avl_node_type structure 21-6
 freeing resources 21-7
 initializing 21-6
 nodes, deleting 21-7
 nodes, inserting 21-6
 nodes, searching for first 21-6
 nodes, searching for next 21-7
 searching 21-7
 walking 21-6
threaded 21-12
 check node is leaf 21-15
 getting greatest node 21-15
 getting next element 21-15
 walking inorder 21-15
wrapped
 freeing resources 21-12
 initializing 21-9
 nodes, inserting 21-9
 nodes, searching for 21-10
 normalizing 21-12
 resetting pointers to start of tree 21-12
 walking 21-10
 avl_delete function 21-7
 avl_get_first function 21-6
 avl_get_last function 21-15
 avl_get_next function 21-7
 avl_get_next_threaded function 21-15
 avl_insert function 21-6
 avl_is_leaf function 21-15
 avl_search function 21-7
 avl_walk function 21-6
 avl_walk_extended function 21-15
AVLDup trees
 deleting a node 21-16
 initializing 21-16
 inserting a node 21-16
 manipulating 21-16
 retrieving first node 21-17

retrieving next node 21-17
searching 21-16
walking 21-17
avldup_delete function 21-16
avldup_get_first function 21-17
avldup_get_next function 21-17
avldup_insert function 21-16
avldup_search function 21-16
avldup_walk function 21-17
AWAKE macro 16-5
example 16-7, 16-8
guidelines for using 16-5

B

background
processes, CEF 23-48
backing store 23-17
backup system 33-1
Banyan VINES
addresses 17-10
Basic Encoding Rules
See BER
bcmf function A-37
bcopy function 5-30
BEEP 25-2
overview 25-2
BER, definition 28-2
BFD 3-31
BIGENDIAN constant 26-13
binary trees
overview 21-1
See also AVL trees, radix trees, RB trees
bit fields
changing minor identifier 3-44
clearing specified bits 3-45
creating 3-44
definition 3-44
deleting 3-45
determining reason process awoken is a changed bit field 3-45
registering a process on 3-44, 3-49
retrieving value of 3-45
setting specified bits 3-45
specifying only one process awoken 3-45
using A-39
bitfield_check function 4-66
bitfield_clear function 4-66
bitfield_clear_many function 4-66
bitfield_clearmask function 4-67
bitfield_destroy function 4-67
bitfield_find_first_clear function 4-66
bitfield_find_first_set function 4-67
bitfield_lock function 4-67

bitfield_set function 4-67
bitfield_set_many function 4-67
bitfield_setmask function 4-67
bitlist
dynamic 4-68
bitlist API 4-68
bitmask_any_bits_in_first_are_set_in_second function 4-67
bleed-through files E-22
Blocks Extensible Exchange Protocol
See BEEP
Boolean Manager 6-2, 6-37
booleans
changing minor identifier 3-43
creating 3-43
definition 3-43
bootstrapping Cisco IOS image
from Flash memory 2-4
from ROM 2-3
over the network 2-3
branching
cloning E-15
introduction E-15
scenarios E-16
collapsing E-18
(table) E-20
assumptions E-18
incremental sync E-4
integration operations E-2
intended audience E-1
life cycle E-7
modeling E-9
porting E-12
advantages and disadvantages E-13
introduction E-12
requirements E-13
porting life cycle
(figure) E-8
reparenting E-10
advantages and disadvantages E-11
introduction E-10
scenarios E-11
sibling branch E-11
stake holders
responsibilities E-26
stakeholders E-26
syncing E-20
advantages and disadvantages E-23
assumptions E-21
bleed-through files E-22
child versions E-22
incremental syncs E-21
reparenting E-21
terminal sync E-21
types of E-3

terminology E-2
types of E-2
broadcasting
 packet duplication 5-28
buffer data, memory pools for 4-17
buffer leaks 20-23
buffer pools
 See packet buffer pools and particles, pools
buffer_stdout function 20-66
buffers
 DMA 23-10
 layer-3 protocol 23-10
 Leaks, tracing 20-21
 public pools 23-9
 See packet buffers
 See also particles
buginf function 17-1, 20-19
buginf() vs printf() 17-2
buses
 Autonomous Switching 23-13
 RSP 23-14
byte order, portability issues 26-2
byte reordering 26-13

C

.c file, registries
 definition 14-13
CACHE
 fast switching
 description 23-11
cache
 fast switching
 populating 23-12
 NetFlow 23-18
 operations 4-83
 Optimum Switching 23-17
caching
 demand-based 23-19
 history 4-85
 issues 4-85
CANA 28-15
CAR
 benefit of enabling CEF and dCEF 23-33
 description 23-33
CASE Services
 manipulating 14-32
case services
 adding (example) 14-34, 14-39
 adding default (example) 14-36
 default function 14-35
 default function, adding (example) 14-36
 defining 14-33, 14-38
 (example) 14-33, 14-39
description 14-3, 14-32, 14-38, G-4
invoking (example) 14-35, 14-40
wrapper functions 14-33, 14-38
CASE_LIST/CASE_LOOP Services
 manipulating 14-54
caution
 description 6-45
caution, description 1-cxiii
cbus
 RSP 23-15
CCN 27-58
CDE B-1
CEF
 adding a new feature
 debugging 23-52
 guidelines 23-49
 overview 23-44
 performance dos and don'ts 23-50
 additional benefits of 23-33
 background processes 23-48
 distributed (dCEF)
 description 23-50
 switching compatibility matrix, CEF globally
 disabled 23-31
 switching compatibility matrix, CEF globally
 enabled 23-31
 switching compatibility matrix, dCEF globally
 enabled 23-32
 example forwarding code 23-48
 feedback, where to send 23-50
 load balancing 23-51
 overview 23-19, 23-20
 resources 23-54
 RSP 23-21
CEF Scalability and Selective Rewrite 23-101
cerno
 definition 20-67
 usage guidelines 20-68
cfgdiff_retcod apply_rollback function 27-95
cfgdiff_retcod do_cfgdiff function 27-92
cfork function
 See process_create function
CFTYPE 27-90
chain.c file 27-64
change_if_input function 5-37
CHANGES file 1-17
CHAR_NUMBER macro 27-14
check_cphog function 3-25
Checking NULL Pointers A-28
Checking the malloc() and getbuffer Return A-30
checkpointing
 defined 24-2
checkqueue function 22-3
chunk manager
 allocating memory chunks 4-34

creating memory chunk 4-30
destroying memory chunks 4-36
guidelines for using 4-29
locking memory chunks 4-36
overview 4-5, 4-29
using A-37
`chunk_create` function 4-30
(example) 4-33
`chunk_destroy` function 4-36
`CHUNK_FLAGS_BIGHEADER` flag 4-31
`CHUNK_FLAGS_DYNAMIC` flag 4-30
`CHUNK_FLAGS_INDEPENDANT` flag 4-30
`CHUNK_FLAGS_INTERRUPT_PROTECTION_OFF` flag 4-32
`CHUNK_FLAGS_MANAGED` flag 4-31
`CHUNK_FLAGS_NOHEADER_FAST` flag 4-32
`CHUNK_FLAGS_NONDATA` flag 4-32
`CHUNK_FLAGS_NONLAZY` flag 4-32
`CHUNK_FLAGS_SIBLING` flag 4-30
`CHUNK_FLAGS_SMALLHEADER` flag 4-31
`CHUNK_FLAGS_TRANSIENT` flag 4-32
`chunk_free` function 4-34
(example) 4-36
`chunk_lock` function 4-36
`chunk_malloc` function 4-34
(example) 4-36
chunks
See memory chunks, chunk manager
Cisco Assigned Numbers Authority 28-15
Cisco Express Forwarding
See CEF 23-19
Cisco IONization 101 Guidelines 1-19
Cisco IOS
bootstrapping image from ROM 2-3
bootstrapping image over network 2-3
Cisco IOS code style issues A-1
Cisco Secure Development Lifecycle-CSDL 11-1
classes, memory pool
See memory pools, classes
classes, region
See regions, classes
`clear ipc stat` 8-51
`clear profile command` C-4
`clear_if_input` function 5-38
CLI 24-48
key bindings 27-2
transient memory 4-41
CLI key bindings 27-2
`cli-output-police@cisco.com` 27-3
clock, system
description 15-2
setting 15-5
clock/calendar
in hardware 15-3
`CLOCK_DIFF_SIGNED` macro 16-10

`CLOCK_DIFF_SIGNED64` macro 16-10
`CLOCK_DIFF_UNSIGNED` macro 16-10
`CLOCK_DIFF_UNSIGNED64` macro 16-10
`clock_epoch` structure 15-1
`clock_epoch_to_timeval` function 15-5
`clock_epoch_to_unix_time` function 15-5
`clock_get_microsecs` function 15-4
`clock_get_time` function 15-3, 15-4
`clock_get_time_exact` function 15-4
`clock_icmp_time` function 15-4
`CLOCK_IN_INTERVAL` macro 16-9
`CLOCK_IN_STARTUP_INTERVAL` macro 16-10
`clock_is_now_valid` function 15-5
`clock_is_probably_valid` function 15-5
`CLOCK_OUTSIDE_INTERVAL` macro 16-9
`clock_set` function 15-5
`clock_set_unix` function 15-5
`clock_time_is_in_summer` function 15-4
`clock_timeval_to_epoch` function 15-5
`clock_timeval_to_unix_time` function 15-5
`clock_timezone_name` function 15-4
`clock_timezone_offset` function 15-4
cloning E-15
particles 5-49
`close_persistence_storage` function 29-41
CNS
Config Agent
commanding the Config Agent 12-7
overview 12-7
registering a callback function 12-7
unregistering a callback function 12-7
DEN 1-7, 12-2
Event Agent
calling `ea_free()` 12-6
clearing the restart callback notification 12-5
considerations 12-6
freeing storage 12-6
notification when the Event Agent shuts down 12-6
reading received event subjects and payloads 12-5
receiving an event 12-4
registering a callback function 12-4
setting the context 12-5
stopping an Event Agent service session 12-6
stopping receiving events 12-6
storing the callback function pointer 12-5
terminating services 12-7
unsubscribing to subjects 12-6
waiting for messages 12-4
watched Booleans 12-5
wild card characters 12-7
Event Agent Services 12-2
overview 12-2
related reading 12-7
sending an event 12-3
starting an event agent session 12-2

- terms 12-1
- cns_config_id_mode_unreg function 12-7
- cns_config_id_set function 12-7
- cns_ea_clear_restart function 12-5
- cns_ea_close function 12-6
- cns_ea_extract_msg function 12-5
- cns_ea_free function 12-6
- cns_ea_open function 12-2
- cns_ea_produce function 12-3
- cns_ea_read function 12-4
- cns_ea_set_callback function 12-4
- cns_ea_set_context function 12-5
- cns_ea_set_restart function 12-5
- cns_ea_setup function 12-5
- cns_ea_subscribe function 12-4
- cns_ea_unsubscribe function 12-6
- cns-ios-sw@cisco.com 12-1
- code
 - columns
 - limit definition A-22
 - Code diffs
 - utility for checking indentation A-23
 - code formatting
 - comments A-26
 - function definitions A-23
 - function prototypes A-23
 - headers A-22
 - if...else statements A-25
 - #include directives A-22
 - indentation A-22
 - parentheses and spaces A-25
 - stubbing out code A-25
 - code organization, description B-4
 - code performance issues
 - algorithms, designing A-36
 - Cisco IOS infrastructure, using A-37
 - data structures, designing A-36
 - GCC optimization A-37
 - instruction-level performance A-37
 - code reliability issues
 - arithmetic overflow, checking for A-30
 - data alignment, checking A-30
 - getbuffer function, checking return A-30
 - malloc function, checking return A-30
 - NULL pointers, checking A-28
 - pointer arithmetic A-30
 - pointers within structures A-30
 - Code Review 37-2
 - code reviews
 - security 30-16
 - codereviewer-ifs@cisco.com 9-1
 - coding
 - error messages, example 19-22
 - Coding Conventions A-1
 - coding conventions A-9, A-17
- bit field instructions A-39
- bit fields in C structures A-39
- C conventions A-17
- coding for multiple operating environments
 - code differences in publicly defined structure field definitions A-50
 - coding for maintainability and safety A-54
 - coding for maximum maintainability A-49
 - coding for performance A-52
 - conditional directives in public header files A-49
 - data-plane performance A-54
 - dealing with OS ports A-53
 - guidelines for application-code developers A-48
 - hiding OS-specific implementation from common code (example) A-51
 - link-time binding (example) A-50
 - motivation for guidelines A-47
 - OS macros in public headers and source files A-49
 - OS-neutral wrapper headers A-53
 - overview A-42
 - targeting underlying OSes A-48
 - targeting underlying OSes (example) A-48
 - task-oriented coding A-54
 - techniques for coding A-54
- coding for security
 - infinite loops and MAX value usage 30-13
 - issues with strcpy(), strcat(), and sprintf() 30-9
 - key storage issues (example) 30-12
 - printf(), snprintf(), and buginf() format strings 30-7
 - strncpy function issues (example) 30-10
 - summary A-42
 - zeroing passwords and keys 30-11
- combinatorics of conditional compilation A-46
- comparing to Kernighan & Ritchie A-17
- conditional compile macros A-47
- conditional inclusions and definitions A-45
- const type qualifiers A-20
- converting signed to unsigned types A-21
- CPU access A-39
- data structure format A-21
- data structures, passing A-21
- #define macros A-4
- dependency management
 - compile-time dependencies A-44
 - description A-43
 - link-time dependencies A-44
 - runtime dependencies A-44
- design issues A-3
- enumerated types A-4
- floating-point operations A-21
- function prototypes A-18
- functions, ordering in a file A-19
- header files A-4
- in VM code 4-79
- incompatibilities between OSes A-47

inline functions A-38
issues with source code across multiple OSes A-43
maintainability of conditionally compiled code A-46
mathematical notations in VM code 4-79
memcpy function A-39
memory, accessing A-39
mixing C and assembly language A-21
modularity domain
 description A-45
multiple dereferencing A-38
multiple operating OSes
 header file relationships A-44
operating system boundary, definition A-42
operating system leaks A-42
performance issues A-35
presentation of code A-21
pretty printing A-21
register declarations A-38
register storage class A-20
reliability issues A-28
repeated code A-38
runtime environment and IOS code A-43
SingleSource component
 description A-44
static storage class A-20
storage A-27
struct copy A-39
transitive header inclusion and conditional compilation A-46
typecasting A-20
variables A-27
volatile keyword A-38
writing code that runs over multiple OSes A-48
writing IOS code for multiple OSes A-43
coding for multiple operating environments A-42
coding for scalability A-40
collapsing branches E-18
Column Width A-22
combinatorics of conditional compilation A-46
Command Length A-22
Command Placement 2-21
commands
 debug parser ambiguity 27-80
 duplicate 27-12
 exiting submode 27-77
 findmerge E-22
 hidden 27-12, 27-74
 internal 27-11, 27-74
 linking 27-69
 ordering 27-68
 parser
 backward compatibility maintaining 27-8
 rcsmerge E-22
scheduler
 examples 3-15
subinterface 27-12
unsupported 27-12
Comments A-26
comments in code
 formatting A-26
 writing A-26
commit procedure
 small features 36-1
Committed Access Rate
 See CAR 23-33
COMP_INC B-1
compatibility queues D-2
compilation, conditional A-4, A-6
compiler-dev@cisco.com 11-2
component
 DDTS, specifying for error message 19-18
components
 of routers 23-6
Conditional Compilation A-6
conditional inclusions A-45
conditional linking A-7
conditionally compiled code
 maintainability A-46
Config Agent
 commanding the Config Agent 12-7
 overview 12-7
 registering a callback function 12-7
 unregistering a callback function 12-7
config checkpointing 27-98
config mode
 error messages 27-36
config replace 27-93
CONFIG_NO_LEVEL_CHANGE 27-78
configuration defaults exposure 27-50
configuration editor in IOS 27-37
configurations 27-8, 27-9
Configure Change Notification 27-60
console
 initialization 7-14
Console Output 20-66
console status block
 See parser, CSB
console_init function 7-3, 7-14, 7-16
Const Qualifier A-20
const type qualifiers, using A-20
controller type 27-90
controllers
 ciscoBus, CxBus, and CyBus 23-13
controlling terminal, setting 3-26
convention in IOS A-27
conventions, coding A-1
Conversion from Signed to Unsigned Types A-21
Coordinated Universal Time 15-3
COPY_TIMESTAMP macro 16-9
core files

analyzing 20-16
debugging CPU exceptions 20-14
generating 20-16
coredump function 7-8
count
 logging 20-43
CPU
 cache memory 4-83
 cache operations 4-83
 maximizing access speed A-39
 sharing A-10
CPU exceptions
 debugging
 overview 20-14
 using core files 20-14
 using GDB 20-17
 using ROM monitor 20-16
 description 20-14
See also exceptions

CPU profiling
 configuring C-3
 CPUHOG mode
 description C-2
 enabling C-5
 enabling C-4
 interrupt mode
 description C-2
 disabling C-5
 enabling C-4
 overhead C-2
 overview C-1
 postprocessing C-5
 profile blocks
 creating C-4
 definition C-1
 deleting C-4
 zeroing C-4
 restarting C-4
 stopping C-4
 task mode
 description C-2
 disabling C-5
 enabling C-4
 using C-3
CPU usage spikes 3-30
CPUHOG condition 3-29
create_watched_queue function
 example 3-28
create_watched_recursive_semaphore() 3-36
critical-priority processes 3-16
cross-platform driver 6-50
crypto_rng() function 3-61
CSB
 See parser, CSB
CSB objects

data variables notes 27-66
csb->nv_command 27-24
CSCso80837
 BK 1-vi, 4-1
CSCso81099
 BK 1-v
CSCso93231
 RB 4-1
CSCsu65721
 KJS 1-vii
CSCsx70756
 BK 1-vii
CSCsx71035
 JP 1-vii
CSCtd34420
 AA 1-vii
CSCtf55129
 ASA 1-viii
CSCth58701
 JR 1-vii
CSCti46419
 JR 1-v
CSCtj08013
 JR 1-vii
CSCtj27179
 JR 1-vi
CSSR 23-101
Curly Braces A-23
current Cisco IOS initiatives 37-1
current_time_source function 15-4
current_time_string function 15-6
cutover
 See switchover

D

Data Alignment A-30
data alignment
 checking A-30
 portability issues 26-6
data blocks 5-20
data size, portability issues 26-6
data structures
 common 9-23
 designing A-36
 formatting A-21
 passing A-21
data_area element, in paktype structure 5-19
data_bytes element, in particletype structure 5-40
data_dequeue function 22-7
data_enqueue function 22-6
data_insertlist function 22-7
data_start element, in particletype structure 5-40
data_walklist function 22-7

datagram_done function 5-26, 5-48
datagramsize element, in paktype structure 5-19
datagramstart element, in paktype structure 5-19
dates, format for printing 17-4
daylight savings time, testing for 15-4
dCEF
 See CEF 23-50
DDTS component
 specifying, error messages 19-18
dead process 3-18
debug
 parser ambiguity 27-80
 debug command 20-18
 debug sanity command 20-14
 debugging 20-2
 CEF feature code 23-52
 cerrno 20-67
 FIB subblocks 23-67
 messages, formatting 17-1
 online documentation, list of links 20-27
Debugging and Error Logging
 enhanced error message log count 20-43
 disabling this feature 20-44
 enabling this feature 20-44
 end user interface 20-44
 using this feature 20-45
 verifying this feature 20-44
Latency Tracer API Functions 20-52
Latency Tracer CLI Commands 20-53
Latency Tracer Implementation 20-54
packet information
 storing 20-46
periodic function 20-49
Receive Latency Tracing 20-45
Receive Trace Facility 20-51
 Latency Tracer API Functions 20-52
 Latency Tracer CLI Commands 20-53
 Latency Tracer Implementation 20-54
 periodic function 20-49
 snapshot display 20-52
structures
 rx_trace_buffer_ 20-47
 rx_trace_efifo_ 20-48
 rx_trace_entry_ 20-47
DECIMAL macro 27-14
decompression
 images
 Warm Upgrade 2-58
default functions
 case services 14-35
 adding (example) 14-36
 fastcase services 14-40
 faststub services 14-51
 ilist services 14-30
 list service 14-27
loop services 14-42
pid_list services 14-32
registries
 description 14-7
 retval services 14-38
 stub services 14-44
 stub_chk services 14-47
default_mac_addr
 MAC address 6-56
DEFAULT_PRIV flag 27-12
#define macros, using A-4
delete_persistence_storage function 29-41
delete_watched_queue function
 example 3-28
delete_watched_rwlock function 3-48
demand paging
 definition 4-82
demand-based caching
 disadvantages of 23-19
DEN
 CNS 1-7, 12-2
dependency relationships
 compile-time dependencies A-44
 description A-43
 link-time dependencies A-44
 runtime dependencies A-44
 semantic dependency
 description A-44
dequeue function 22-4
 (example) 22-5
dereferencing, multiple, in code A-38
DestroyRBTTree function 21-5
dev_macros.h file 6-55
dev_master.h 6-52
dev_object_t structure 6-50, 6-51
dev_util.h file 6-52
Dev-Object Model 6-50
DHCP
 overview 25-4
dhcpc_params structure 25-5
direct queues
 (figure) 22-1
 adding items 22-3, 22-5
 description 22-1
 initializing 22-2
 protected 22-5
 (example) 22-6
 removing items 22-4, 22-6
 unprotected 22-3
 (examples) 22-4
directories
 file systems 9-21
disable_preemption() 3-36
disposition 23-95
distributed CEF (dCEF)

See CEF 23-50
distributed environment
access lists in 23-34
FIB subblocks 23-63
GSR 23-29
switching
 VIP, dCEF and distributed switching 23-22
 switching compatibility matrix 23-30
`dllob_remove` function 22-17
`dllobj_add` function 22-15
`dllobj_add_after` function 22-15
`dllobj_add_before` function 22-16
`dllobj_add_to_front` function 22-16
`dllobj_in_list` function 22-16
`dllobj_init` function 22-16
`dllobj_insert_ordered` function 22-16
`dllobj_nth` function 22-16
`dllobj_read` function 22-16
DMA
 description 23-10
DNS 25-7
 overview
 submitting queries 25-8
`DNSRES_DONE` flag 25-8
documentation
 debugging, links to 20-27
 Doxygen F-1
Domain Name Service
 See DNS
`domain_submit_domain_query` function 25-8
`domain_submit_query` function 25-8
domains
 DNS lookup 25-8
doubly linked lists
 (examples) 22-9, 22-13
 adding elements 22-15
 adding items 22-9, 22-11
 contents, displaying 22-12
 creating 22-10
 deleting elements 22-17
 description 22-2
 destroying 22-13
 functions 22-15
 initializing 22-16
 inserting elements 22-16
list action functions
 changing behaviors 22-12
 default behaviors 22-12
 retrieving behaviors 22-12
list_element data structure 22-11
`LIST_FLAGS_AUTOMATIC` flag 22-11
`LIST_FLAGS_INTERRUPT_SAFE` flag 22-11
moving items 22-11
reading elements 22-16
removing items 22-9, 22-11

searching for elements 22-16
See also list manager
Doxygen instructions F-1
DPM
 defined 24-3
DRAM
 CEF 23-20
 RSP 23-15
drivers
 cross-platform 6-50
 Ethernet interface 29-23
 interface
 registration (example) 29-9
dynamic bitlists 4-68
dynamic pools 5-7
dynamic region manager
 creating a freeregionlist 4-38
 enabling transient memory feature 4-38
 overview 4-38

E

`edisms` function
 See `process_wait_for_event` function
editors
 configuration 27-37
`EDP` 37-2
EIM
 description
 interface driver 29-23
`ELAPSED_TIME` macro 16-9
`ELAPSED_TIME64` macro 16-9
email
 alias archives, searching 1-18
Embedded Resource Manager 4-43
Embedded Syslog Manager
 See ESM
`enable_mib_persistence` function 29-41
`enable_preemption()` 3-36
`ENCAPBYTES` packet data padding 5-20
encapsulation
 layer-2 23-11
 layer-3 23-10
end command 27-75
Enhanced 16-31
Enhanced Timer Wheel 16-31
enqueue function 22-4
 (example) 22-5
Ensuring Correct Results A-20
entities, IPC
 definition 8-4
ENTITY-MIB
 entity objects
 adding 29-37

deleting 29-37
looking up 29-38
Enumerated Types and #defines A-4
enumerated types, using A-4
EOL node 27-95
epoch
 clock_epoch structure 15-1
 definition 15-1
errmsg function 19-1, 19-3, 19-20
 (example) 19-20
errmsg_ext function 19-20, 19-21
error logging
 cerrno 20-67
 enhanced 20-43
Error Message Decoder 19-2
error message explanation
 creating 19-12
error message facility
 defining 19-7
error messages
 action, specifying a recommended 19-14
 coding example 19-22
 component, specifying a DDTS 19-18
 components of msg_xxx.c file 19-5
 config mode 27-36
 creating explanation 19-12
 defining 19-4, 19-7
 defining in error message file 19-6
 facility, defining 19-7
 file
 description 19-4
 generating 19-20
 macros in msg_xxx.c 19-5
 mnemonic 19-5
 severity values (table) 19-9
 specifying
 DDTS component 19-18
 recommended action 19-14
 submitting for review 19-22
TAC engineer, identifying information for 19-18
 testing 19-19
errvarmsg function 19-20
ESM
 overview 20-66
ESR 10000
 architecture 23-28
 description 23-27
Ethernet Interface Manager
 See EIM
Event Agent
 calling ea_free() 12-6
 clearing the restart callback notification 12-5
 considerations 12-6
 freeing storage 12-6
 notification when the Event Agent shuts down 12-6
reading received event subjects and payloads 12-5
receiving an event 12-4
registering a callback function 12-4
sending an event 12-3
 (example) 12-3
setting the context 12-5
starting a session 12-2
stopping an Event Agent service session 12-6
stopping receiving events 12-6
storing the callback function pointer 12-5
terminating services 12-7
unsubscribing to subjects 12-6
waiting for messages 12-4
watched Booleans 12-5
wild card characters 12-7
Event Agent Services 12-2
EVENT_TRACE_BASIC_INSTANCE macro 20-31, 20-32
event_trace_dump function 20-38
event_trace_insert_node function 20-39
event_trace_one_shot function 20-34
event_trace_print function 20-35
event_trace_print_all function 20-35
event-trace mechanism 20-2
Examples
 enhanced error message log count 20-45
examples
 coding for multiple operating environments
 hiding OS-specific implementation from common code A-51
 link-time binding A-50
 coding for multiple operating systems
 targeting underlying OSes A-48
 driver that implements address filter function
 vectors 6-20
Event Agent
 sending an event 12-3
parser
 commands, linking 27-69
scheduler
 commands 3-15
security
 key storage issues 30-12
strncpy function
 security issues 30-10
exception handler
 causing exceptions 18-4
 overview 18-1
 registering 18-2, 18-3
 signals (table) 18-1
exceptions
 initialization
 (example) 7-12
 overview 18-1
 See also exception handler

exit command 27-75
exit_all_config_submode function 27-75
exit_config_submode function 27-75
extern storage class, using A-27

F

facdef macro 19-5

fast switching
 MCI/CiscoBus 23-35
 overview 23-11
 writing code 23-35
fast_malloc function 4-19
FASTCASE Services
 manipulating 14-38
fastcase services 14-40
FASTSTUB 14-47
FASTSTUB Services
 manipulating 14-47
faststub services 14-51
Feature as a Subsystem A-8
fenced timers
 See managed timers

FIB
 adjacency table 23-46
 function tables
 (example) 23-59
 IDBs 23-44

subblocks
 control encode routine 23-68
 creating, initializing, and deleting 23-61
 data structure, contents 23-60
 data structure, types 23-57
 debugging 23-67
 distributed environment 23-63
 example implementation 23-57
 function table 23-58
 managing 23-60
 overview 23-55
 resources 23-69
 using 23-63
technology
 how it works 23-45

fibswsb_ft_structure 23-58

file systems
 classes 9-2
 data structures 9-23
 directories 9-21
 hooks 9-26
 NVRAM 9-28
 simple
 (examples) 9-5
 timestamps 9-22
 types 9-2

file_reg.mk 14-19
files
 error message 19-4
FILES_REG 14-19
filesystems
 initialization 7-30
findmerge command E-22
Flexible Packet Matching feature 20-25
flo_sync@cisco.com 1-16
floating point 11-35
Floating-Point Operations A-21
floating-point operations, in code A-21
flow
 NetFlow
 description 23-18
 flow tag 23-18
FOR_ALL_HWIDBS_IN_LIST macro 6-34
FOR_ALL_SWIDBS macro 6-5
FOR_ALL_SWIDBS_IN_LIST macro 6-34
Format of Data Structures A-21
Format Specifier for a Pointer A-30
format_time function 15-6
format-scrub@cisco.com 17-1
Formatting Block Comments A-26
formatting strings
 debugging messages 17-1
 time 17-4
 user command output 17-1
forwarding
 See switching 23-15
Forwarding Information Base
 See FIB 23-20
Forwarding Information Base (CEF)
 description 23-20
forwarding processors
 SSO
 system platform architecture 24-9
FP
 defined 24-3
FPU 11-36
fragmentation
 memory allocation 4-38
free function 4-18, 4-23, 4-24
 (example) 4-25
free lists
 overview 4-15
 sizes
 adding 4-15
 default 4-15, 4-26
 setting 4-26
freeregionlist
 creating 4-38
 freeing memory to add to 4-40
 initializing 4-38
 (figure) 4-38

Function Prototypes A-18

function table

FIB subblock 23-58

functions

definitions, spaces with A-23

ordering in a file A-19

prototypes

spaces with A-23

using A-18

G

Garbage Detector 4-59

Garbage Detector feature 20-23, 20-25, 20-27

GCC

optimizing A-38

using A-17

GDB

analyzing core files 20-16

kernel mode 20-17

process mode 20-18

using to debug CPU exceptions 20-17

GENERAL_CHAR_NUMBER macro 27-14, 27-16

GENERAL_KEYWORD macro 27-10

GENERAL_NUMBER macro 27-13, 27-15

octal parsing 27-16

generating

error messages 19-20

geo_sync@cisco.com 1-16

get or set the state, configuration, or statistics fields for a particular interface 6-39

get_interrupt_level function 2-22

GET_NONZERO_TIMESTAMP macro 16-9

GET_TIMESTAMP macro 16-9

(example) 16-10

GET_TIMESTAMP32 macro 16-9

getbuffer function 5-25, 5-26

GETLONG function 26-14

GETOBJ 27-65

getparticle function 5-45

GETSHORT macro 26-14

Gigabit Switch Router

See GSR

Gnu CC compiler

See GCC

Greenwich Mean Time 15-3

GSR

dCEF 23-29

switching path 23-13

three stages of packet path 23-30

H

.h file, registries

definition 14-13

HA

assumptions 24-6

definition 24-5

hardware

architectural assumptions 24-8

automatic switchover, a fault on the active RP 24-11

automatic switchover, active RP declared

"dead" 24-11

automatic switchover, criteria-based 24-12

line card behavior for SSO 24-10

manual switchover, CLI invoked 24-11

RP switchover behavior 24-10

RP switchover conditions for SSO 24-11

SSO system platform architecture 24-8

system model 24-8

trial switchover, CLI invoked 24-11

introduction 24-1

platforms

Cisco 10000 ESR, description 24-58

Cisco 10000 ESR, forwarding and control architecture 24-59

Cisco 10000 ESR, taxonomy (table) 24-59

Cisco 12000 GSR, forwarding and control architecture 24-63

Cisco 12000 GSR, overview 24-62

Cisco 12000 GSR, taxonomy (table) 24-62

Cisco 7500, forwarding and control architecture 24-61

Cisco 7500, overview 24-60

Cisco 7500, taxonomy (table) 24-60

overview 24-58

target platform set 24-7

requirements

dCEF 24-7

IOS images 24-7

NSF 24-7

overview 24-6

route processors 24-6

target platform architectures 24-58

terminology 24-2

handle subsystem 4-86

hardware

early

Multibus 23-35

platforms

HA SSO arch. system model 24-8

PXF 23-25

SSE 23-13

Hardware Session API 23-69, 23-73

Header Files A-4

header files, bracketing with conditional compilation statements A-4
headers, standard A-22
heaps
 managing with memory pools 4-15
 memory pools for 4-17
HEXADECIMAL macro 27-14
HEXDIGIT macro 27-14
HIDDEN 27-75
hierarchy, memory
 See regions, hierarchy
high-priority processes 3-17
Hot ICE 27-40
HSU
 defined 24-3
hung process 3-18
HW Abstraction API 23-70
hwidb_bm_allocate function 6-37
hwidb_bm_assign function 6-37
hwidb_bm_clear function 6-37
hwidb_bm_set function 6-37
hwidb_bm_test function 6-37
HWSB_FAST define constant 6-28

I

ICD 27-90
ID Manager 4-86
IDB
 FIB IDBs 23-44
 idb_add_hwidb_to_list function 6-34
 idb_add_hwsb function 6-21
 idb_add_swidb_to_list function 6-34
 idb_add_swsb function 6-21
 idb_bm_allocate function 6-37
 idb_bm_assign function 6-37
 idb_bm_clear function 6-37
 idb_bm_set function 6-37
 idb_bm_test function 6-37
 idb_board_encap function 6-38
 idb_create_list function 6-34
 idb_delete_hwsb function 6-22
 idb_dequeue_from_output function 6-39
 idb_destroy_list function 6-35
 idb_enqueue function 6-5
 idb_enqueue_full function 6-46
 idb_erase_subif function 6-46
 idb_for_all_on_hwlist function 6-38
 idb_for_all_on_swlist function 6-38
 idb_get_hwsb function 6-22
 idb_get_swsb function 6-22
 idb_hw_get_only_swidb function 6-46
 idb_hw_state_config function 6-39
 idb_is_* functions 6-38

 idb_pak_vencap function 6-39
 idb_proto_counter_allocate function 6-9
 idb_proto_counter_decrement function 6-10
 idb_proto_counter_get function 6-10
 idb_proto_counter_increment function 6-10
 idb_proto_counter_reset function 6-10
 idb_proto_counter_set function 6-10
 idb_queue_for_output function 6-39
 idb_remove_from_list function 6-35
 idb_start_output function 6-39
 idb_subif_init function 6-47
 idb_identity
 definition 6-3
IDBs
 adding to global queue 6-47
 address filter function vectors 6-19
 associate with a packet buffer 5-36
 ATM IDB recycling
 description 6-8
 enabling layer 2 subblock reusability 6-9
 removing data structures 6-8
 creating 6-3
 data structure shrinking
 overview 6-35
 deleting 6-5
 enqueueing 6-46
 freeing 6-6
 hardware
 definition 6-1
 deleting 6-5
 unlinking 6-5
 index number 6-3
 initializing subblock IDB 6-47
 interface locking mechanism 6-47
 introduction 6-3
 iterating list 6-5
 iterating over private list 6-34
 iterating over private list safely 6-34
 linking 6-5
 linking to router interfaces 6-5
 linktype enum 6-44
 making visible 6-47
 nextsub subinterface list feature 6-46
 only one swidb exists 6-46
 oqueue vector 6-39
 oqueue_dequeue vector 6-39
 packets
 dequeueing 6-39
 encapsulating 6-38, 6-39
 queuing 6-39
 transmitting 6-39
 private lists
 adding IDBs 6-34
 applying function vector and argument to IDB 6-38
 creating 6-34

deleting 6-35
description 6-33
removing IDB 6-35
removing from nextsub queue 6-46
removing from swidbList 6-46
reusing 6-6
software
 definition 6-1
 deleting 6-5
 unlinking 6-5
 soutput vector 6-39
Subblock Modularity
 API changes 6-29
 changes 6-25
 converting existing subblocks 6-32
 current subblock use 6-24
 data structures 6-26
 dynamically allocated boolean flags 6-36
 issues 6-31
 overview 6-23
subblocks
 adding to IDB 6-21
 deleting from IDB 6-22
 dynamic, description 6-21
 obtaining pointer to hardware IDB 6-22
 obtaining pointer to software IDB 6-22
subinterfaces
 freeing 6-6
 unlinking 6-5
testing interface properties 6-38
unit number 6-3
unlinking 6-5
IDECIMAL macro 27-14
idle queue 3-8
If...Else Statements A-25
if...else statements A-25
IF-Index-definition 6-3
IF-MIB
 tables 29-30
ifRcvAddressTable, IF-MIB table 29-30
ifs_create function 9-22
ifs_ioctl function 9-11
ifs_iopen function 9-11
ifs_istat function 9-11
ifs_lseek function 9-14
ifs_rename function 9-15
ifStackTable, IF-MIB table 29-30
ifTable, IF-MIB table 29-30
IFTYPE 27-89
ifType_get function 29-42
ifXTable, IF-MIB table 29-30
ILIST Services
 manipulating 14-27
ilist services
 default function 14-30
images
 decompressing and transferring control 2-58
imposition 23-95
#include directives, using A-22
incremental sync E-4
indentation, in code A-22
index tables
 creating 22-23
 deleting elements 22-23
 finding elements 22-23, 22-24
 finding first vacancy 22-23
 freeing all tree nodes 22-23
 inserting elements 22-24
indexed object lists 22-17
 adding an element 22-19
 cleaning up 22-20
 finding size of 22-19
 getting the nth element 22-19
 initializing an index object 22-18
 performing a linear search 22-19
 removing an element 22-19
 searching for element 22-19
india-mem-team@cisco.com 4-1
indirect queues
 adding items 22-6
 description 22-2
 examples 22-7
 figure 22-2
 initializing 22-2
 iterating over 22-7
 removing items 22-7
 size, changing 22-7
 walking 22-7
indxobj_add function 22-19
indxobj_cleanup function 22-20
indxobj_find_position function 22-19
indxobj_get_nth_element function 22-19
indxobj_init function 22-18
indxobj_linear_search function 22-19
indxobj_remove function 22-19
indxobj_search function 22-19
indxobj_search_next function 22-19
indxobj_size function 22-19
indxtbl_create function 22-23
indxtbl_delete_element function 22-23
indxtbl_free function 22-23
indxtbl_get_first_element function 22-23
indxtbl_get_first_vacancy function 22-23
indxtbl_get_last_element function 22-24
indxtbl_get_next_element function 22-24
indxtbl_get_next_vacancy function 22-24
indxtbl_get_nth_element function 22-24
indxtbl_get_prev_element function 22-24
indxtbl_insert_element function 22-24
informs, definition 28-3

initialization
 init_process function
 final tasks 7-36
 platform-specific 7-23
 exception 7-10
 fundamental 7-4
 line 7-33
 overview 7-2
 strings 7-37
 (table) 7-37
 strings(example) 7-38
 values 7-40
 (example) 7-41
 (table) 7-40
system
 basic 2-2
 by ROM monitor 2-2
 description 2-1 to 2-20
 fundamental (figure) 2-18
 of Cisco IOS image 2-19
inline assembler 26-12
inline functions, using A-38
INPUT Q 23-10
input queue of an interface 5-37
 definition 5-3, G-11
input_getbuffer function 5-37
insqueue function 22-4
integers
 security vulnerabilities 30-13
 unsigned 30-15
Intelligent Config Diff 27-90
interactive CLI 27-99
interest-fib mailing list 23-50
interest-ifs@cisco.com 9-1
interest-ipc@cisco.com 8-2
interest-mem@cisco.com 4-1
interest-mib@cisco.com 28-1
interest-os@cisco.com 3-1, 5-5, 6-1, 13-1, 15-1, 16-1,
 18-1, 21-1, 22-1, 37-1
interest-os-logging@cisco.com 20-1
interest-os-registry@cisco.com 14-1, 14-12
interest-parser@cisco.com B-1
interest-socket@cisco.com 10-1
interest-syslog@cisco.com 20-1
interface descriptor blocks
 See IDBs
interface submodes
 limiting searches 27-78
interface types 27-89
INTERFACE_KEYWORD macro 27-10
interfaces
 initialization 7-23
 stack relationship 29-19
Internet Network Management Framework,
 description 28-2
Internet Network Management Framework, description
 (figure) 28-2
interprocess communications
 See IPCs
interrupt
 in process switching 23-10
interrupt_environment 16-13
interrupts
 getting levels 2-22
 raising levels 2-23
 resetting levels 2-24
 setting levels 2-24
INTF_RANGE_COND_INTF_SUBMODE 27-77
INUMBER macro 27-13
invoke_safe_str_constraint_handler function 11-8
IOCTAL macro 27-14
IOMEM
 packet data buffers 5-19, 5-21
iomem function 7-8
ION 1-19, 14-8, 14-22
IOS caching
 history 4-85
IOS CLI web pages 27-1
IOS Config Rollback 27-93
IOS Memory Leaks 20-27
ios_platforms@cisco.com 7-1, 26-1
ios_timeval structure 15-1
 converting from 15-5
 converting to 15-5
ios-crypto@cisco.com 3-65
ios-errmsg-review@cisco.com 19-1, 19-2
ios-ha-dev@cisco.com 24-1
ios-images-ready-for-pilot@cisco.com 34-1
ios-infra-security@cisco.com 30-1
ios-scalers@cisco.com 32-1
ios-security@cisco.com 30-1
ios-style-guide@cisco.com A-1, B-1, D-1
ios-switch-coders@cisco.com 23-1
ios-white-hats@cisco.com 30-1
IP
 Cisco Express Forwarding (CEF) 23-20
IPADDR macro 27-66
IPC
 get port notifications 8-61
 notifications 8-61
 port notifications 8-61
 RX port-level events
 notification 8-62
 TX port-level events
 notification 8-62
IPC Master 8-58
 overview 8-58
ipc_add_named_seat function 8-23
ipc_close_port function 8-31
ipc_create_named_port function 8-26

(example) 8-42
ipc_get_message function 8-33
ipc_get_pak_message function 8-33
ipc_get_seat function 8-23
ipc_locate_port function 8-30
ipc_message_header structure 8-6
ipc_open_port function 8-26
ipc_open_port_by_name function 8-27
 (example) 8-42
ipc_process_raw_pak function 8-33
ipc_register_port function 8-26
ipc_remove_port function 8-31
ipc_reset_seat function 8-24
ipc_resync_seat function 8-24
ipc_return_message function 8-35
ipc_send_message function 8-33
 (example) 8-43
ipc_send_message_blocked function 8-33
 (example) 8-42
ipc_send_rpc function 8-38
ipc_send_rpc_blocked function 8-38
ipc_send_rpc_reply function 8-38
ipc_send_rpc_reply_blocked function 8-38
ipc_set_rpc_timeout function 8-38
ipc-dev-team@cisco.com 8-2
IPCS
 implementing on RSP platform 8-43
IPCs
 applications, creating 8-41
 blocking send for reliable messages
 (figure) 8-12
 blocking send for RPC messages
 (figure) 8-16
 CiscoBus driver 8-43
 close port message flow
 (figure) 8-31
 communication
 overview 8-3
 communication environments
 (table) 8-2
 congestion status notification capability 8-38
 current receive IPC message processing through seat to
 client
 (figure) 8-37
 current tx message processing through seat
 (figure) 8-34
 entities, definition 8-4
 IOS-IPC process 8-18
 (figure) 8-18
 IPC Master 8-58
 ipc_message_header structure 8-6
 lookup message
 (figure) 8-29
 manipulating message retransmission table 8-32
 manipulating port table 8-24
 manipulating seat table 8-22
 message format 8-6
 message format-version 1
 (figure) 8-8
 message retransmission table
 description 8-32
 entries 8-32
 messages
 definition 8-4
 dispatching received packets 8-33
 format 8-6
 format (figure) 8-7
 retrieving header 8-33
 returning 8-35
 sending 8-33, 8-42, 8-43
 multicast ports, definition 8-4
 name registration message flow
 (figure) 8-26
 non-blocking send for reliable message with overlapping
 (figure) 8-15
 non-blocking send for reliable message without
 overlapping
 (figure) 8-14
 non-blocking send for RPC messaging
 (figure) 8-17
 non-blocking send for unreliable messaging
 (figure) 8-10
 non-blocking send for unreliable with notification
 (figure) 8-11
 on RSP platform 8-43 to 8-50
 open port message flow for local port
 (figure) 8-29
 open port message for remote port
 (figure) 8-30
 operational environment
 loosely coupled 8-3
 networked 8-3
 tightly coupled 8-2
 unispace 8-2
 overview 8-3
 port naming services 8-5
 port table
 description 8-24
 entries 8-24
 port_info structure 8-42
 ports
 closing 8-31
 creating by name 8-26
 creating by name, example 8-41
 definition 8-4
 finding by name 8-30
 identifiers, definition 8-4
 names, definition 8-4
 names, reserved 8-6
 naming conventions 8-5

naming syntax 8-5
opening by identifier 8-26
opening by name 8-26
registering by name 8-26
removing 8-31
receive IPC messages
 overview 8-35
references 8-62
reliable messaging 8-12
remove port message flow
 (figure) 8-32
reserved port names
 (table) 8-6
RPC messaging 8-16
RPCs
 setting timeout period 8-38
 simulating asynchronous response 8-38
 simulating send 8-38
 simulating synchronous response 8-38
RX port-level events
 notification 8-62
seat manager, definition 8-4
seat table
 description 8-23
 entries 8-23
seats
 creating 8-23
 definition 8-4
 resetting 8-24
 retrieving from seat table 8-23
sending messages
 overview 8-32
sequence numbers, resetting 8-24
services
 (figure) 8-2
services, overview 8-2
simulating RPCs 8-38
terminology 8-4
TX port-level events
 notification 8-62
unreliable messaging 8-9
unreliable messaging with notification 8-10
useful debugging commands 8-51
writing an IPC application 8-41
zone manager, definition 8-5
zones, definition 8-4

IPv6
 addresses 17-12
is_valid_pc function 4-13
isGeneralInfoGroupOnly function 29-41
ISR 3-31
item_desc_init function 6-41
 resetting items 6-41
item_list_malloc function 6-40
itemlist_reset function 6-41

J
jitter
 timers 16-2

K
keys
 security issues 30-11
KEYWORD macro 27-10
KEYWORD_ID macro 27-10
KEYWORD_MM macro 27-10
KEYWORD_NOWS macro 27-10
KEYWORD_OPTWS macro 27-10
knox-dev@cisco.com 2-1

L
L2 Hardware Switching API 23-69, 23-74
L2F 23-91
L2TP 23-81
L2TPv2 23-91
L2TPv3 23-86
L2VPN 23-1
L2X 23-91
Layer 2 Tunneling Protocol 23-81
LBL config checkpointing 27-98
leaf timers
 See managed timers
least recently used, definition 4-83
libregistry.a 14-9
line cards
 behavior in order to support SSO 24-10
 SSO
 system platform architecture 24-8
line-by-line (LBL) checkpointing 27-98
link points
 creating 27-70
 description 27-70
 displaying 27-71
 exit link points, creating 27-72
 linking command to 27-72
 registering with parser 27-71
LINK_POINT 2-21
LINK_TRANS macro 27-70
linked lists
 See doubly linked lists, list manager, queues, singly linked lists
links to debugging-related online documentation 20-27
list manager
 description 22-2, 22-10
 See also doubly linked lists

LIST services
manipulating 14-25
list services
adding
(example) 14-26
adding (example) 14-28
default function 14-27
defining 14-25, 14-27
(example) 14-25, 14-28
description 14-3, 14-27, G-13
invoking
(example) 14-26
invoking (example) 14-30
wrapper functions 14-25, 14-27
list_create function 22-10
(example) 22-13, 22-14
list_dequeue function 22-11
(example) 22-14
list_destroy function 22-13
(example) 22-15
list_element data structure 22-11
list_enqueue function
(example) 22-13
example 22-14
LIST_FLAG_AUTOMATIC flag 22-11
LIST_FLAGS_INTERRUPT_SAFE flag 22-11
list_get_action function 22-12
list_get_info function 22-13
list_insert function 22-11
list_move function 22-11
(example) 22-14
list_remove function 22-11
(example) 22-14, 22-15
list_requeue function 22-11
list_set_action function 22-12
list_set_automatic function 22-11
list_set_info function 22-12
(example) 22-13
list_set_interrupt_safe function 22-11
lists
indexed object 22-17
LITTLEENDIAN constant 26-13
load balancing
along switching paths 23-32
CEF 23-51
load_mib function 29-41
load_persistence_data function 29-40
LOCAL areas
virtual memory 4-76
locks
read/write 3-48
log count
enhanced 20-43
LOG_DEFAULT_QUEUESIZE 7-40
logging facility

enhanced 20-43
LOOP Services
manipulating 14-40
loop services
adding (example) 14-41
default function 14-42
defining 14-40
(example) 14-41
description 14-3, 14-40, G-13
invoking (example) 14-42
wrapper functions 14-40
loopback detection 4-68
low-priority processes 3-17
LRU
managing IDs 4-89
lw_insert function 22-9
(example) 22-9
lw_remove function 22-9
(example) 22-9

M

MAC address
default_mac_addr 6-56
macros
conditional compilation usage A-47
make_idb_visible function 6-47
malloc function 4-19, 4-23
example 4-23
malloc_aligned function 4-19
malloc_iomem function 4-19
malloc_iomem_aligned_function 4-19
malloc_lite function 4-43, 4-58
malloc_named function 4-19
malloc_named_aligned function 4-19
malloc_named_fast function 4-19
malloc_named_iomem function 4-19
malloc_named_iomem_aligned function 4-19
malloc_named_pcimem function 4-19
malloc_named_pcimem_aligned function 4-19
malloc_pcimem function 4-19
managed bit fields
See bit fields
managed boolean
See booleans
managed queues
types of 3-41
managed timers 16-1
(example) 16-17
context value
description 16-11
extended, retrieving 16-16
extended, setting 16-16
initializing 16-13

modifying 16-14
returning 16-14
definition 16-11
fenced timers
 creating 16-16
 definition 16-16
 returning 16-17
initializing 16-13
interrupt routines, using 16-11
jitter 16-2
leaf timers
 changing to parent timers 16-17
 definition 16-11
 delay, increasing 16-14
 expiration, setting 16-14
 initializing 16-13
 starting 16-14
 stopping 16-15
linking to other timer trees 16-16
`mgd_timer` structure 16-12
operation 16-11
overview 16-11
parent timers
 changing to leaf timers 16-17
 definition 16-11
 determining address of first child 16-17
 determining address of next sibling 16-17
 initializing 16-13
 stopping 16-15
processes, registering on timer 16-13
state, determining 16-13, 16-15
stopping 16-15
type value
 description 16-11
 returning 16-13
 setting 16-13
unlinking from other timer trees 16-16
walking timer tree 16-17
`managed_chunk_create` function 4-37
`managed_chunk_free` function 4-37
`managed_chunk_malloc` function 4-37
`managed_chunk_process` function 4-37
`managed_chunk_queue` 4-37
Management Information Base
 See MIBs
maximum rate of traffic
 CAR 23-33
maximum task time
 description 3-29
MCEF 30-19
MCI/ciscoBus
 architecture 23-35
`mem_lock` function 4-24
 (example) 4-25
`mem_unlock` function 4-25
`mem-busters@cisco.com` 4-1
`memcmp` function A-37
`memcmp_s` function 11-9
`memcpy_s` function 11-9
MEMD
 See SRAM
`memmove_s` function 11-9
memory
 access types 4-83
 allocating 4-18
 aligned 4-19
 buffer data 4-19
 example 4-23
 failures 4-21
 fast 4-19
 fragmentation 4-38
 free 4-23
 `freeregionlist` 4-38
 heap 4-19
 interleaving transient and static memory 4-38
 return value 4-20
 table 4-19
 transient 4-20, 4-38
 typecasting 4-20
 unaligned 4-19
 allocation 7-28
 cache
 NetFlow 23-18
 DRAM
 CEF 23-20
 RSP 23-15
 hierarchy
 See regions, hierarchy
 initialization
 (example) 7-8
 locking 4-24
 (example) 4-24
 MMU 4-73
 platform memory initialization
 `freeregionlist` 4-38
 process
 RSP 23-15
 returning 4-24
 RSP 23-14
 secure mode
 comparing strings 11-9
 comparing values in 11-9
 copying region of 11-9
 initializing area of 11-9
 locating substring 11-9
 moving blocks 11-9
 zeroing bytes 11-9
 shared 4-89
 shared memory 23-41
 SRAM (MEMD)

RSP 23-14
transient
 API functions 4-40
 virtual, overview of Cisco IOS 4-73
memory check-interval command 4-43
memory chunks
 (figure) 4-6
 allocating 4-34
 (example) 4-36
 return value 4-34
 typecasting 4-34
 chunk manager 4-29, A-37
 creating 4-30
 (example) 4-33
 destroying 4-36
 locking 4-36
 returning 4-34
 returning (example) 4-36
memory debug leaks reclaim command 20-24
memory io critical rising command 20-23
Memory Leak Detector feature 20-23, 20-27
memory malloc-list-use-malloc command 4-43
memory management
 managed chunks 4-1, 4-5, 4-36
 managed_chunk_create function 4-37
 managed_chunk_free function 4-37
 managed_chunk_malloc function 4-37
 managed_chunk_process function 4-37
 managed_chunk_queue 4-37
memory management unit
 See MMU
memory pool manager 4-5
memory pools
 (figure) 4-6
 adding regions to 4-16
 aliases
 declaring 4-17
 (example) 4-18
 overview 4-17
 allocating memory
 aligned 4-19
 buffer data 4-19
 example 4-23
 failures 4-21
 fast 4-19
 heap 4-19
 return values 4-20
 table 4-19
 transient 4-20
 typecasting 4-20
 unaligned 4-19
 alternate 4-18
 creating 4-18
 creating (example) 4-18
 buffer data 4-17
bytes free 4-28
bytes used 4-28
classes
 aliasable 4-17
 mandatory 4-17
 MEMPOOL_CLASS_IOMEM flag 4-17
 MEMPOOL_CLASS_LOCAL flag 4-17
 MEMPOOL_CLASS_TRANSIENT flag 4-17
 setting 4-17
 table 4-17
 creating 4-16, 4-37
 (example) 4-16
 definition 4-5
free lists
 sizes, adding 4-15
 (example) 4-26
 sizes, default 4-15, 4-26
 sizes, setting 4-26
freeing memory 4-24
heaps 4-15, 4-17
initializing 4-39
low memory
 setting threshold 4-27
 specifying action to take 4-27
low-water mark 4-27
memory pool manager 4-15
overview 4-15
returning memory 4-24
searching for 4-27
 (example) 4-28
static usage
 (figure) 4-39
statistics, retrieving 4-28
threshold, low
 dropping below 4-27
 setting 4-27
total bytes 4-28
transient 4-17, 4-38
 allocating aligned memory 4-41
transient usage
 (figure) 4-39
memory pools for 4-17
Memory Resource Owner 20-25
memory resource policy 20-25
memory statistics history table command 4-43
memory traceback recording
 description 20-65
memory try-malloc-lite command 4-43
memory validate-checksum command 4-43
mempool_add_alias_pool function 4-17
 (example) 4-18
mempool_add_alternate_pool function 4-18
 example 4-18
mempool_add_free_list function 4-26
 (example) 4-26

mempool_add_region function 4-16
MEMPOOL_CLASS_IOMEM flag 4-17
MEMPOOL_CLASS_LOCAL flag 4-17
MEMPOOL_CLASS_TRANSIENT flag 4-17
mempool_create function 4-16
 (example) 4-16
mempool_find_by_addr function 4-27
 (example) 4-28
mempool_find_by_class function 4-27
 (example) 4-28
mempool_get_free_bytes function 4-28
mempool_get_total_bytes function 4-28
mempool_get_used_bytes function 4-28
mempool_is_empty function 4-27
mempool_is_low function 4-27
mempool_malloc function 4-19
mempool_malloc_aligned function 4-19
mempool_set_fragment_threshold function 4-27
mempool_set_low_threshold function 4-27
mempools
 See memory pools
memset_s function 11-9
memzero_s function 11-9
message facility
 enhanced 20-43
message log count
 enhanced 20-43
message retransmission table, IPC
 description 8-32
 entries 8-32
messages
 determining if queued 3-49
 determining if simple message is queued 3-49
IPC
 definition 8-4
 dispatching received packets 8-33
 retrieving header 8-33
 returning 8-35
 sending 8-33, 8-42, 8-43
 registering a process for notification 3-49
 sending at interrupt level 3-48
 threshold 20-51
MFI 23-95
mgd_timer structure 16-12
mgd_timer_additional_context function 16-16
mgd_timer_change_to_leaf function 16-17
mgd_timer_change_to_parent function 16-17
mgd_timer_context function 16-14
 example 16-19
mgd_timer_delink function 16-16
mgd_timer_exp_time function 16-15
mgd_timer_expired function 16-15
 example 16-18
MGD_TIMER_EXTENDED macro 16-16
mgd_timer_first_child function 16-17
mgd_timer_first_expired function 16-15
 example 16-18
mgd_timer_first_fenced function 16-17
mgd_timer_first_running function 16-15
mgd_timer_init_leaf function 16-13
 (example) 16-18
mgd_timer_init_parent function 16-13
 (example) 16-18
mgd_timer_initialized function 16-13
mgd_timer_left_sleeping function 16-15
mgd_timer_left_sleeping64 function 16-15
mgd_timer_link function 16-16
mgd_timer_next_running function 16-17
mgd_timer_running function 16-15
mgd_timer_running_and_sleeping function 16-15
mgd_timer_set_additional_context function 16-16
mgd_timer_set_context function 16-14
mgd_timer_set_exptime function 16-14
mgd_timer_set_fenced function 16-16
mgd_timer_set_soonest function 16-14
mgd_timer_set_type function 16-13
mgd_timer_start function 16-14
 (example) 16-18
mgd_timer_start_jittered function
 (example) 16-18
 example 16-19
 prototype 16-14
mgd_timer_start64 function 16-14
mgd_timer_stop function 4-23, 16-15
 example 16-19
mgd_timer_type function 16-13
 example 16-18
mgd_timer_update function 16-14
 (example) 16-19
mgd_timer_update_jittered function 16-14
MIB compiler
 examples 28-31
 functon 28-30
 invoking 28-21, 28-29
 location 28-21
 mibcomp.perl script
 invoking 28-29
 options (table) 28-30
 mosy 28-21
 output files 28-31
 overview 28-20
 SMICng 28-21
 update-mibs.pl script, invoking 28-21
MIB data
 persisting 29-40
MIB Persistence
 closing persistence storage 29-41
 deleting underlying file 29-41
 enabling/disabling persistence 29-41
 loading persisted MIB data from NVRAM 29-41

loading persistent data 29-40
opening underlying file in non-volatile storage
 area 29-40
 writing data to NVRAM 29-40
`mibcomp.perl` script, invoking 28-29
`mib-dev@cisco.com` 29-1
MIB-II support 1-11
`mib-police@cisco.com` 29-1
MIBs
 agent implementation 28-4
 branch numbers, assigning 28-15
 branch points, description 28-4
 Cisco Assigned Numbers Authority (CANA) 28-15
 compilers
 See MIB compiler
 compiling
 examples 28-31
 location of generated files 28-28
 makefile rules 28-29
 overview 28-28
 which groups to compile 28-28
 which MIBs to compile 28-28
 definition 28-2
 design considerations 28-11
 alerts 28-13
 assigned numbers 28-15
 checking existing MIBs 28-14
 Cisco MIB nomenclature 28-15
 information flow control 28-13
 informs 28-13
 MIB compliance 28-14
 MIB content 28-12
 MIB conventions 28-14
 MIB organization 28-14
 MIB police 28-22
 MIB template 28-16
 notifications 28-12
 phases 28-12
 polling 28-13
 reliable delivery 28-13
 support 28-22
 traps 28-13
 writing conventions for MIBs 28-15
IF-MIB API
 adding support to service points 29-32
 deregistering a sublayer 29-35
 external files 29-31
 IANAifType Textual Convention 29-31
 internal files 29-31
 link up/down trap support 29-35
 registering a sublayer 29-34
 sample implementation 29-35
 subiabtype data structure 1-11, 29-31
 tables 29-30
 informs, definition 28-13
instance identifiers 28-5
`k_get` routines 28-34
`k_set` routines 28-35
leaf objects, description 28-4
life cycle 28-10
maintaining 28-43
maintaining (example) 28-44
modifying 28-43
modifying (example) 28-44
modularity, observing 28-33
new, creating 28-25
nomenclature, Cisco 28-15
objects
 adding to MIB 28-44
 definition 28-4
 deleting from MIB 28-45
 identifiers 28-5
 identifiers (figure) 28-6
 implementing 28-33
 `k_get` routines 28-34
 `k_set` routines 28-35
 operations 28-43
 testing 28-40
 overview 28-3
 persistence across reloads 29-40
 phases in life cycle 28-10
 police, MIB 28-22
 proprietary, description 28-4
 releasing 28-43
 standard, description 28-4
 subinterface
 IANAifType Textual Convention 29-31
 template, Cisco 28-16
 testing
 notifications 28-41
 notifications, tools to use 28-42
 objects 28-40
 overview 28-40
 tools to use, command-line 28-41
 tools to use, X Windows 28-41
 top-level identifier, determining 28-25
 traps 29-43
 traps, definition 28-13
 version control 28-44
`miss-manners@cisco.com` A-1
Mixing C and Assembly Language A-21
MLP
 adding serial interfaces 29-19
MMU
 definition 4-80
 virtual memory requirement 4-73
mnemonic
 error messages 19-5
more
 changing automore's header in midstream 17-3

conditionally asking for permission to do more output 17-4
disabling "---MORE---" processing 17-3
finding if user has quit automore 17-3
turning on automatic "---MORE---" processing 17-2

MPLS 23-81
msclock 16-2
msclock variable 16-2
msg_def_tac_details macro 19-5
msg_xxx.c file
 components 19-5
 msg_ipx.c 19-4
msgdef function 19-1
msgdef macro 19-5
 parameters (table) 19-8
msgdef macros 19-19
msgdef_ddts_component macro 19-5
msgdef_explanation macro 19-5
msgdef_limit macro 19-5
msgdef_recommended_action macro 19-5
msgdef_required_info macro 19-5
msgdef_section function 19-5
msgsym function 19-20

MTBF
 defined 24-4, G-15

MTTR
 defined 24-4, G-15

Multibus 23-35
multicast ports, IPC, defintion 8-4
multicasting
 packet duplication 5-28
multiple dereferencing, in code A-38
Multi-Protocol Label Switching 23-81
mwiedman-dev-group@cisco.com 9-1

N

N_ITEM_LIST macro 6-40
NetFlow Switching
 description 23-18
 Feature Acceleration 23-50
 performance 23-19
network interrupt throttling 2-21
Network Time Protocol 15-3
 See NTP
network_start element, in paktype structure 5-20
networks
 HA
 requirements 24-6
 hiding internal 30-20
 NMI routine
 timers 16-2
no mem try-malloc-lite command 4-18
no profile command C-4

NO_OR_DEFAULT macro 27-20
non-volatile storage
 initialization 7-21
normal-priority processes 3-17
NRP 30-19
NSF
 defined 24-4
NSP 30-19
NTP 15-3
NUMBER macro 27-13, 27-15, 27-66
NUMBER_OCT flag 27-16
NUMBER_STRICT flag (deprecating) 27-15
numbers, parsing 27-13
nv_init function 7-21, 7-22
 (example) 7-22
nv_write function
 prototype 17-2
NVGEN 2-20, 2-21
 configuration defaults exposure 27-50
 subsystem sequencing 2-20
NVGEN config checkpointing 27-98
NVGEN enhancement project 27-98
NVRAM
 API 9-28
 functions 9-28
 initialization 7-22

O

OBJ 27-65
 non-zero during Show 27-9
OBJ(type, N) variable 27-9
object
 indexed lists 22-17
objects
 See MIBs, objects
Obscure C Features A-20
OCE 23-101
OCT_NO_ZERO_OK flag 27-16
OCT_ZERO_OK flag 27-16
OCTAL macro 27-14
OCTAL_NOT_STRICT flag 27-16
OID
 See MIBs, objects
ONE_ITEM_LIST macro 6-40
 (example) 6-40
oneshot, definition 3-41, 3-44, 3-49
open_persistence_storage function 29-40
operating systems
 code differences in publicly defined structure field
 definitions A-50
 coding for maintainability and safety A-54
 conditional directives in public headers A-49
 dealing with OS ports A-53

guidelines for application-code developers A-48
header file relationships A-44
hiding OS-specific implementation from common code
(example) A-51
incompatibilities between OSes A-47
issues with using source code A-43
leaks A-42
link-time binding (example) A-50
OS macros in public headers and source files A-49
OS-neutral wrapper headers A-53
targeting underlying OSes A-48
targeting underlying OSes (example) A-48
task-oriented coding A-54
techniques for coding A-54
writing code that runs over multiple OSes A-48
writing IOS code A-43
Optimum Switching
description 23-17
steps 23-17
OPTIONAL_KEYWORD 27-10
oqueue vector 6-39
oqueue_dequeue vector 6-39
Order of Functions within a File A-19
ORDER_BYTE_LONG macro 26-13
ORDER_BYTE_SHORT macro 26-13
Ordering Commands 27-68
os-infra-team@cisco.com 5-5, 17-1, 21-1, 22-1
out-of-band 30-19
Output Chain Element 23-96
Output Interpreter 19-2
OUTPUT Q 23-11
overview
 HA 24-1
 SSO
 implementation 24-17

P

p_dequeue function 22-6
(example) 22-6
p_enqueue function 22-5
(example) 22-6
p_requeue function 22-5
p_unqueue function 22-6
p_unqueueuenext function 22-6
packet buffer
 centering a frame 5-31
packet buffer caches
 creating 5-33, 5-34
 filling 5-33, 5-34
 getting a buffer 5-34
 removing buffers 5-34
packet buffer pools
 assigning a fallback pool 5-24

best fit 5-24
caches
 creating 5-33
 filling 5-33, 5-34
 getting a buffer 5-34
 removing buffers 5-34
description 5-21
dynamic 5-7
fallback pool 5-24
group number 5-7
item size 5-7, 5-24
private 5-21
 allocating buffers 5-26
 creating 5-23, 5-24
 description 5-21
 fallback pool 5-24
 group number 5-23
public 5-21
 allocating buffers 5-25, 5-26
 creating 5-22, 5-23
 description 5-21
 filling 5-22
 group number 5-21, 5-22
 returning buffers 5-26
packet buffers
 adding to interface input queue 5-37
 allocating 5-25, 5-26
 associate with an IDB 5-36
 associating with an input interface 5-37
 centering 5-31
 cloning 5-31, 5-32
 copying 5-31
 buffer and context 5-30
 buffer only 5-31, 5-32
 recentering 5-31
 copying entire data area 5-30
 datagramsize element, in paktype structure 5-19
 datagramstart element, in paktype structure 5-19
 duplicating 5-29, 5-31, 5-32
 buffer and context 5-30
 buffer only 5-31, 5-32
 memory corruption 5-30
 recentering 5-31
 duplicating entire data area 5-30
 duplicating given size of data area 5-31, 5-32
 duplicating to newly obtained buffer 5-32
 fragmentation and reassembly 5-28
 headers, definition 5-19
 initialization 7-18
 interface input queue 5-37
 interface input queue count 5-3, 5-37, G-11
 IOMEM 5-19
 locking 5-26
 moving to another input interface 5-37
 network_start element in paktype structure 5-20

packet data blocks 5-19
 memory organization 5-19
 memory organization (figure) 5-20
 packet design (figure) 5-18
 paktype structure 5-19
 paktype structure 5-19
POOL_PUBLIC_DUPLICATE flag 5-32
pools 5-7
reference count 5-26, 5-27
removing from an input interface 5-38
returning 5-26
reusing 5-28
size, increasing 5-33
structure 5-18
truncating during duplication 5-30
unlocking 5-26
packet data blocks 5-19
 definition 5-19
ENCAPBYTES padding 5-20
memory organization 5-19
memory organization (figure) 5-20
TRAILBYTES padding 5-20
packet information
 storing 20-46
packet reparenting 5-48
packets
 dequeuing 6-39
 encapsulating 6-38, 6-39
 queuing 6-39
 transmitting 6-39
page fault
 definition 4-82
page out and page in 4-76
paging
 definition 4-81
pak_center function 5-31
pak_clip function 5-32
pak_coalesce function 5-47
pak_common_cleanup function 5-28
pak_copy function 5-30
pak_copy_and_recenter function 5-31
pak_dequeue function 22-7
 (example) 22-8
pak_duplicate function 5-31, 5-32
pak_enqueue function 22-6
 (example) 22-8
pak_grow function 5-33
pak_inqueue function 22-6
 example 22-8
pak_lock macro 5-26
pak_pool_create function 5-22, 5-23
 (example) 5-24
pak_pool_create_cache function 5-33, 5-34
pak_pool_find_by_size function 5-24
pak_requeue function 22-7
 (example) 22-8
pak_reset function 5-28
pak_unqueue function 22-7
 (example) 22-8
pakDB 5-19
pakqueue_resize function 22-7
paktype structure 5-19
 data_area 5-19
 datagramsize 5-19
 datagramstart 5-19
 network_start 5-20
 PAKBASE_DEF macro 5-19
PAM MBOX 30-19
 interface 30-19
PARAMS macro 27-17
PARAMS_KEYONLY macro 27-17
parent timers
 See managed timers
parentheses, spaces around A-25
parser 27-50
 alternate mode 27-75
 chain.c file 27-64
 CLI interface functions 9-29
 CLI keyboard shortcuts
 key bindings
 parser 27-2
 command parser analysis 27-80
commands
 backward compatibility maintaining 27-8
 duplicate 27-12
 exiting submode 27-77
 hidden 27-12, 27-74
 hiding using macros 27-26
 internal 27-11, 27-74
 not supported checking 27-33
 subinterface 27-12
 unsupported 27-12
 unsupported, blocking 27-24
 unsupported, coding best practices 27-24
 unsupported, definition 27-24
 unsupported, use of documentation 27-36
commands, linking 27-69
commands, ordering 27-68
config mode
 error messages 27-36
config rollback
 overview 27-93
configuration defaults exposure
 API 27-50
 overview 27-50
configuration deprecation process 27-8
configuration deprecation process after 18 months 27-9
console status block 27-4
CSB objects 27-65
 data variables notes 27-66

debugging parser ambiguity 27-80
DEFAULT_PRIV flag 27-12
developer's resources 27-1
end command 27-75
error messages 27-36
EXEC mode
 debugging parser ambiguity 27-80
exit command 27-75
exit functions 27-75
exiting all config submodes 27-75
exiting config submode 27-75
exiting submode 27-77
finite state machine 27-2
GETOBJ 27-65
GETOBJ functionality 27-66
help string format 27-6
images
 supported 27-25
implicit exit to parent mode 27-75, 27-76
information sources 27-1
key bindings 27-2
keyword tokens
 parsing 27-10
 parsing (example) 27-12
 privilege levels 27-11
 transition diagram (figure) 27-13
keyword-number tokens, parsing 27-17
keyword-number tokens, parsing (example) 27-17
limiting searches in submodes 27-75, 27-78
link points
 (list) 27-68
 creating 27-70
 description 27-70
 displaying 27-71
 exit, creating 27-72
 linking commands to 27-72
 registering 27-71
model 27-2
modes
 adding 27-78
 adding (example) 27-79
 aliases, adding 27-79
no commands, processing (example) 27-23
node
 definition 27-2
nodes
 transition structure 27-6
nonvolatile output, generating
 csb->nv_command 27-24
 description 27-4, 27-24
number tokens, parsing 27-13
number tokens, parsing (example) 27-16
nv_add_check_default() 27-50
OBJ 27-65
 non-zero during Show 27-9
OBJ(type, N) variable 27-9
optional keywords, parsing 27-18
overview 27-2
parse graph
 definition 27-3
parse tree
 traversing 27-4
parse trees
 linking 27-64
 linking (example) 27-64
 NVGEN 27-68
 traversing 27-4
 traversing (example) 27-5
parse trees, building 27-7
parser component
 IOS Parser Group 27-1
parser development group
 contact info 27-1
parser trees, troubleshooting 27-1
PRIV_CHANGE_DISALLOWED flag 27-12
PRIV_CHANGE_DISALLOWED flag 27-12
PRIV_DISTILLED flag 27-11
PRIV_DUPLICATE flag 27-12
PRIV_HIDDEN flag 27-12, 27-35
PRIV_INTERACTIVE flag 27-11
PRIV_INTERNAL flag 27-11
PRIV_MAX flag 27-11
PRIV_MIN flag 27-11
PRIV_NO_SYNALC flag 27-12
PRIV_NOHELP flag 27-12
PRIV_NONVGEN flag 27-12
PRIV_NULL flag 27-11
PRIV_OPR flag 27-11
PRIV_ROOT flag 27-11
PRIV_SUBIF flag 27-12
PRIV_UNSUPPORTED flag 27-12, 27-35
PRIV_USER flag 27-11
PRIV_USER_HIDDEN flag 27-12
registries
 supported/unsupported features 27-28
submode exit 27-75
subsystems
 supporting features 27-25
syntax checking 27-54
tokens
 parser ambiguity, parsing 27-80
 transition structure, definition 27-6
Parser Chains 27-7
parser config cache interface command 27-98
parser function
 Recevie Trace Facility 20-52
Parser Police Manifesto 27-1
Parser Return Codes 27-37
parser, deprecating CLI syntax 27-8
parser, deprecating CLI syntax after 18 months 27-9

parser_add_command_list function 27-72
parser_add_commands function 27-69
parser_add_link_exit function 27-72
parser_add_link_point function 27-71
parser_add_mode function 27-78
parser-dev@cisco.com 27-3
parser-police@cisco.com 27-3
parser-questions@cisco.com 27-3
parsing
 system startup 24-22
particle pools
 See particles, pools
particle_dequeue function 5-47
particle_enqueue function 5-46
particle_get_refcount function 5-46
particle_lock function 5-46
particle_pool_create function 5-45
particle_pool_create_cache function 5-45
particle_retbuffer 5-48
particle_retbuffer function 5-46
particles 5-39
 appending particle to chain 5-46
 chains 5-41
 cloning 5-49
 description 5-5, 5-39
 overview 1-3
 paktype structure 5-41
 particletype structure 5-40
 pool size 5-7
 pools
 caches, creating 5-45
 creating 5-45
 description 5-39, 5-41
 locking a particle 5-46
 obtaining particle from 5-45
 returning entire particle chain 5-46
 returning particle chain and packet header 5-46
 returning particle to 5-46
 removing head particle from chain 5-47
 reparenting 5-48
 structure 5-40
 (figure) 5-40
particletype structure
 data_bytes 5-40
 data_start 5-40
 description 5-40
pas_allocate_fspak function 7-27
pas_buffer_mempool function 7-28
pas_init_fspak_pool function 7-28
pas_instance_init_common function 7-27
pas_interface_fallback_pool function 7-28
pas_interface_header_pool function 7-28
pas_is_channelized function 7-27
pas_platform_if_final_init function 7-28
pas_slots_init function 7-27
Passing Structures A-21
passive timers 16-1
 in the future
 timers
 passive in the future 16-3
 watching 16-8
passive timers in the future
 delay
 adding to timestamp 16-7
 increasing 16-4, 16-6
 subtracting from timestamp 16-7
 expiration, setting 16-4
 starting 16-4
states
 awake 16-3
 determining (table) 16-5
 expired 16-3
 figure 16-3
 running 16-3
 sleeping 16-3
 stopped 16-2
 unexpired 16-3
 stopping 16-4
passive timers in the past
 (example) 16-10
 definition 16-8
 determining current timestamp 16-9
 elapsed time, determining 16-9
 testing whether time is within range 16-9
timestamps
 copying 16-9
 copying atomically 16-9
 current, obtaining 16-9
 testing whether time is within bounds 16-10
passwords
 security issues 30-11
performance
 CEF
 dos and don'ts 23-50
 NetFlow Switching 23-19
 performance analysis
 webpage A-36
 performance, designing code for A-35
permanent pool items 5-8
persistence support for MIBs 29-41
Phase Containment 37-1
physical address
 definition 4-82
 description 4-75
PID
 definition 3-8
 how assigned 3-20
 retrieving 3-26
 values assigned 3-20
PID_LIST Services

manipulating 14-30
pid_list services
 adding (example) 14-32
 default function 14-32
 defining 14-31
 defining (example) 14-31
 description 14-3, 14-30, G-19
 invoking (example) 14-32
 wrapper functions 14-31
pkt_scheduling@cisco.com 3-1
platform
 dependency on in fast switching 23-11
platform memory
 initializing 4-38
platform_buffer_init function 7-18, 7-20
platform_exception_init function 7-3, 7-10, 7-12, 7-13
platform_file_init function 7-30, 7-31
platform_get_string function 7-37, 7-39
 (example) 7-38
platform_get_value function 7-40, 7-41
 (example) 7-41
platform_interface_init function 7-4, 7-23, 7-25
platform_line_init function 7-4, 7-32, 7-33, 7-34
platform_main function 7-3, 7-4, 7-6
platform_memory_init function 7-3, 7-9
platform_nvvar_support function 7-23
PLATFORM_OTHER option 2-57
platform_pcimempool function 7-29
platform_redirect_console_output function 20-66
platform_reenable_console_output function 20-66
PLATFORM_STRING_BOOTROM_OR_BOOTFLASH
 flag 7-38
PLATFORM_STRING_DEFAULT_HOSTNAME
 flag 7-37
PLATFORM_STRING_HARDWARE_REVISION
 flag 7-37
PLATFORM_STRING_HARDWARE_REWORK
 flag 7-37
PLATFORM_STRING_HARDWARE_SERIAL
 flag 7-37
PLATFORM_STRING_HTML_DEFAULT_PATH
 flag 7-38
PLATFORM_STRING_HTML_FIRMWARE_VERSION
 flag 7-38
PLATFORM_STRING_HTML_HOME_FLAG
 flag 7-38
PLATFORM_STRING_LAST_RESET flag 7-38
PLATFORM_STRING_MIDPLANE_VERSION
 flag 7-38
PLATFORM_STRING_NOM_DU_JOUR flag 7-37
PLATFORM_STRING_PCMCIA_CONTROLLER
 flag 7-38
PLATFORM_STRING_PROCESSOR flag 7-37
PLATFORM_STRING_PROCESSOR_REVISION
 flag 7-37
PLATFORM_STRING_PROMPT_PREFIX flag 7-38
PLATFORM_STRING_PROMPT_SUFFIX flag 7-38
PLATFORM_STRING_VENDOR flag 7-37
PLATFORM_VALUE_CPU_TYPE flag 7-40
PLATFORM_VALUE_FAMILY_TYPE flag 7-40
PLATFORM_VALUE_FEATURE_SET flag 7-40
PLATFORM_VALUE_HARDWARE_REVISION
 flag 7-40
PLATFORM_VALUE_HARDWARE_SERIAL
 flag 7-40
PLATFORM_VALUE_LOG_BUFFER_SIZE flag 7-40
PLATFORM_VALUE_LOG_MAX_MESSAGES
 flag 7-40
PLATFORM_VALUE_REFRESH_TIME flag 7-40
PLATFORM_VALUE_SERVICE_CONFIG flag 7-40
PLATFORM_VALUE_VENDOR flag 7-40
platform_verify_config function 7-36
platform-dependent code 7-2
platform-generic code 7-2
platform-independent code 7-2
platforms
 HA
 requirements 24-6
 platform-specific code 7-2
 Pointer Arithmetic A-30
 pointer arithmetic, performing A-30
 Pointers within Structures A-30
 policies
 applied only to first packet in a flow 23-18
pool caches
 adding to pool 5-15
 description 5-14
 filling 5-16
 removing from pool 5-17
 structure 5-15
 (figure) 5-15
 vectors
 (table) 5-16
 vectors, prototypes 5-16
pool_adjust function 5-22
pool_adjust_cache function 5-16
 example 5-34
pool_cache_vector structure 5-16
pool_create_cache function 5-17
pool_create_group function 5-23, 5-24
pool_dequeue_cache function 5-34
pool_destroy function 5-17
pool_enqueue_cache function 5-34
pool_getbuffer function 5-26
pool_getparticle function 5-45
POOL_GROUP_PUBLIC flag 5-21, 5-22
pool_item_vectors structure 5-12
POOL_PUBLIC_DUPLICATE flag 5-32
pool_set_cache_threshold function 5-17
pools

caches 5-14
 adding to pool 5-15
 filling 5-16
interface reenabling 5-17
removing 5-17
structure 5-15
 (figure) 5-15
threshold callback function 5-17
throttling solution 5-17
vectors
 (table) 5-16
vectors, prototypes 5-16
creating 5-8
description 5-6
dynamic 5-7
filling 5-12
group number 5-7
inserting into a list 5-7
permanent items 5-8
pooltype structure 5-6
private
 creating 5-8
public
 creating 5-8
 filling 5-12
public buffer 23-9
size of pool items 5-7
static 5-7
structure 5-6
 (figure) 5-7
temporary items 5-8
vectors
 (table) 5-11
vectors, prototypes 5-12
See also packet buffer pools and particles, pools
pooltype structure 5-6
port adapters
 initialization 7-25
port names, IPC, definition 8-4
port table, IPC
 description 8-24
 entries 8-24
port_info structure 8-42
ports, IPC
 closing 8-31
 creating by name 8-26
 (example) 8-41
 definition 8-4
 finding by name 8-30
 identifier, definition 8-4
 multicast, definition 8-4
 names, reserved 8-6
 naming
 conventions 8-5
 syntax 8-5
opening
 by identifier 8-26
 by name 8-26
registering by name 8-26
removing 8-31
POSIX standard functions 1-ix, 11-16
posix_printf() function 11-22
PRC 27-37
PRC error codes 27-42
PRC Phase III 27-40, 27-41
PRC_ALREADY_ENABLED 27-43
PRC_BAD_SUBCOMMAND 27-42
PRC_BUSY 27-43
prc_change_mode_t 27-43
prc_change_type_t 27-43
PRC_CHECKPOINT_FAILURE 27-43
PRC_CONFIG_CHANGE 27-43
PRC_CONFIG_CHANGE_NO_NVGEN 27-43
PRC_CONFIG_NO_CHANGE 27-43
PRC_DEFERRED 27-44
PRC_EAGAIN 27-42
PRC_ENOMEM 27-42
prc_error_code_t 27-42
PRC_FAILURE_PERMANENT 27-43
PRC_FAILURE_RETRY 27-43
prc_failure_type_t 27-43
PRC_GEN_FAILURE 27-42
PRC_GENERAL_FAILURE 27-42
PRC_IMMEDIATE 27-43
PRC_INITIALIZATION_FAILED 27-43
PRC_INVALID 27-42
PRC_INVALID_CMD_SEQ 27-43
PRC_INVALID_PARAMETER 27-43
PRC_IO_FAILURE 27-43
PRC_MSG_SEND_ERROR 27-43
PRC_NO_MEMORY 27-43
PRC_NOT_ENABLED 27-43
PRC_PARSE_ERROR_AMBIG 27-42
PRC_PARSE_ERROR_CFG_DENIED 27-42
PRC_PARSE_ERROR_INVALID 27-42
PRC_PARSE_ERROR_LOCKED 27-42
PRC_PARSE_ERROR_NOAUTH 27-42
PRC_PARSE_ERROR_NOERR 27-42
PRC_PARSE_ERROR_NOMATCH 27-42
PRC_PARSE_ERROR_NOMATCH_ALIAS 27-42
PRC_PARSE_ERROR_NOMEM 27-42
PRC_PARSE_ERROR_REMOTE_FAILURE 27-42
PRC_PARSE_ERROR_STBY_NOMATCH 27-43
PRC_PARSE_ERROR_UNKNOWN 27-42
PRC_REBOOT 27-43
PRC_SUCCESS 27-42
PRC_TIMING_UNKNOWN 27-44
PRC_UNINITIALIZED 27-42
PRC_UNSUPPORTED_FEATURE 27-43
preemptive process 3-31

prepaging
 definition 4-82

prep-commit command 19-19

Primary Timer Wheel Data Structures 16-27

printf function 15-6, 17-1

printf() vs buginf() 17-2

printing strings
 See strings, formatting

PRIV_CHANGE_DISALLOWED flag 27-12

PRIV_CONF 27-11

PRIV_DISTILLED flag 27-11

PRIV_DUPLICATE flag 27-12

PRIV_HIDDEN flag 27-12

PRIV_INTERACTIVE flag 27-11

PRIV_INTERNAL flag 27-11

PRIV_MAX flag 27-11

PRIV_MIN flag 27-11

PRIV_NO_SYNALC flag 27-12

PRIV_NOHELP flag 27-12

PRIV_NONVGEN flag 27-12

PRIV_NULL flag 27-11

PRIV_OPR flag 27-11

PRIV_ROOT flag 27-11

PRIV_SUBIF flag 27-12

PRIV_UNSUPPORTED flag 27-12

PRIV_USER flag 27-11

PRIV_USER_HIDDEN flag 27-12

private buffer pools 5-21

private lists
 See IDBs, private

process accounting 3-29
 inaccuracy 3-30
 SAVE_INTR_START_TIME macro 3-30
 UPDATE_INTR_CPU_USAGE macro 3-30

process ID
 See PID

process memory 23-15

process number 3-8

process quantum
 description 3-21, 3-29

process restart command 27-98

process switching
 overview 23-10

process_create function 3-19, 3-26
 example 3-20

process_create_preemptive() 3-36

process_exists function 3-27

process_get_analyze function 3-26

process_get_arg_num function 3-26

process_get_arg_ptr function 3-26

process_get_crashblock function 3-26

process_get_name function 3-26

process_get_pid function 3-26

process_get_priority function 3-26

process_get_profile function 3-26

process_get_runtime function 3-26

process_get_socket function 3-26

process_get_stacksize function 3-26

process_get_starttime function 3-26

process_get_ttynum function 3-26

process_get_ttysoc function 3-26

process_get_wakeup function 3-52

process_get_wakeup_reasons function 3-26, 3-52

process_is_high_priority function D-11

process_is_ok function 3-27

process_kill function 3-28

process_may_suspend function 3-22

process_rwlock_rdlock function 3-48

process_rwlock_rdlock_timed function 3-48

process_rwlock_unlock function 3-48

process_rwlock_wrlock function 3-48

process_rwlock_wrlock_timed function 3-48

process_safe_sleep_for function 3-6, 3-57

process_safe_sleep_on_timer() function 3-57

process_safe_sleep_periodic function 3-6, 3-57

process_safe_sleep_until() function 3-57

process_safe_suspend function 3-6, 3-26, 3-57

process_safe_wait_for_event() function 3-58

process_set_all_profiles function 3-26

process_set_analyze function 3-26

process_set_arg_num function 3-20, 3-26
 example 3-20

process_set_arg_ptr function 3-20, 3-26
 example 3-20

process_set_crashblock function 3-26

process_set_debug function 3-26

process_set_name function 3-26

process_set_priority function 3-26
 See process_create function

process_set_profile function 3-26

process_set_socket function 3-26

process_set_timing function 3-26

process_set_ttynum function 3-20, 3-26
 example 3-20

process_set_ttysoc function 3-20, 3-26

process_set_wakeup_reasons function 3-26

process_suspend function 3-21

process_suspend_if_req function 3-24

process_suspend_allowed function 3-25

process_time_exceeded function 3-25

process_wait_for_event function 3-40

process_wait_for_event_timed function 3-40

process_wait_on_system_config function 3-55

process_wait_on_system_init function 3-55

process_wake_timer_wheels function 16-32

process_wakeup function 3-51

process_wakeup_w_reason function 3-51

process_watch_mgd_timer function 16-13
 example 3-28

process_watch_queue function

example 3-28
process_watch_timer_wheel function 16-32
process_would_suspend function 3-25
processes
 accounting 3-29
 fix inaccuracies 3-30
 analyzing post-mortem 3-26
 arguments
 passing 3-26
 retrieving 3-26
 background 23-48
 checking if ok 3-27
 controlling terminal, setting 3-26
 CPU usage spikes 3-30
 CPU utilization 3-29
 CPUHOG condition 3-29
 creating 3-19, D-8
 creating (example) 3-20
 debugging,setting 3-26
 delaying
 example 3-55
 description 3-16
 destroying 3-27
 SIGEXIT 3-28
 determining reasons process awoke 3-52
 determining whether PID exists 3-27
 directly waking up 3-51
 disabling wait, sleep and suspends 3-56
 guaranteed quantum 3-22
 ionized 4-92
 killing 3-27
 messages
 retrieving for 3-49
 sending to 3-49
 modifying start time,setting 3-26
 moving between queues 3-7, 3-9
 name
 retrieving 3-26
 setting 3-26
PID
 how assigned 3-20
 retrieving 3-26
 values assigned 3-20
POSIX 4-93
priority 3-16
 critical 3-16
 definition 3-16
 high 3-17
 low 3-17
 normal 3-17
 retrieving 3-26
 setting 3-26
process watchdog 3-29
profiles, setting 3-26
registering on a timer 3-50, 3-51
 relinquishing the CPU 3-22, 3-23
 running time, retrieving 3-26
 socket structure,setting 3-26
 stack size, retrieving 3-26
 starting 3-20
 starting time, retrieving 3-26
state
 dead 3-18
 hung 3-18
 ready to run 3-17
 running 3-17
 sleeping (absolute time) 3-17
 sleeping (interval) 3-17
 sleeping (managed timer) 3-18
 sleeping (periodic) 3-18
 table 3-17
 waiting for event 3-17
stopping
 description 3-27
 when system crashes 3-26
suspending 3-20
 conditionally 3-22, 3-23
 determining if able to 3-25
 exceeded "process quantum" 3-25
 exceeded maximum task time 3-25
 for specified amount of time 3-50
 for specified time interval 3-55
 ready to run or exceeded "process quantum" 3-25
 unconditionally 3-21
 until absolute time 3-50
 until asynchronous event occurs 3-53, 3-55
 until managed timer expires 3-50
 suspending (table) 3-21
 suspending if used "process quantum" 3-24
 suspending safely
 for specified amount of time 3-50
 unconditionally 3-26
 until absolute time 3-50
 until managed timer expires 3-50
THIS_PROCESS flag 3-27
waking up
 determining next reason 3-52
waking up, classes of events allowed 3-26
waking up, reasons for 3-51
waking up, supplying a reason 3-51
waking up, using DIRECT_EVENT 3-51
watchdog 3-31
profile blocks
 creating C-4
 definition C-1
 deleting C-4
 zeroing C-4
profile command C-4, C-5
profile hogs commands C-5
profile start command C-4

profile stop command C-4
profile task command C-4
profilers@cisco.com C-1
profiling, CPU
 See CPU profiling
Prototyping Functions A-19
provision 23-84
provisioning model 23-81
psec
 troubleshooting stack corruption 20-74
pseudo-preemption 3-31
public buffer pools 5-21
PUTLONG macro 26-14
PUTSHORT macro 26-14
PW 23-99

Q

queue
 transmit
 backing store 23-17
queue_init function 22-2
 (example) 22-5, 22-6
QUEUEEMPTY macro 22-3
QUEUEFULL macro 22-3
QUEUEFULL_RESERVE macro 22-3
queues
 adding data pointer to 3-42
 available space, determining 22-3
 changing minor identifier 3-41
 critical, operation D-3
 critical-priority, operation (figure) D-4
 definition 3-41
 description 22-1
 determining whether item is on queue 22-3
 direct
 See direct queues
 empty, determining whether 22-3
 full, determining whether 22-3
 high-priority, operation D-4
 high-priority, operation (figure) D-5
 indirect
 See indirect queues
 initializing 22-2
 locating first item on 3-42
 low-priority operation (figure) D-7
 low-priority, operation D-5
 medium-priority operation (figure) D-7
 medium-priority, operation D-5
 moving processes between 3-9
 number of items on queue, determining 22-3
 operation, description 3-9, D-2
 operation, description (figure) D-3
 priority (figure) D-2

registering a process on 3-41
specifying only one process awoken per enqueue 3-41
types of 3-7, 3-41
See also doubly linked lists, list manager, singly linked lists
queues, compatibility
 See compatibility queues
queues, idle
 See idle queue
QUEUESIZE macro 22-3

R

radix trees
 initializing 21-17
 nodes
 deleting 21-18
 inserting 21-17
 searching for 21-18
 overview 21-2
 parent nodes, marking 21-18
 walking 21-17
raise_interrupt_level function 2-23, A-37
rand() function 3-61
Random Number Generation Facility 3-59
Random Number Generators
 crypto functions in all images 3-61
 crypto functions in only crypto images 3-61
 reading on the web 3-65
random_fill function 3-64
 example 3-64
random_gen function 3-63
 example 3-63
random_gen() suite of functions 3-60
random_gen_32bit function 3-63
 example 3-63
random_gen_32bit_context function 3-63
 example 3-63
random_gen_context function 3-63
 example 3-63
random_init function 3-62
 example 3-62
random_init_context function
 example 3-62
rate parameters
 CAR 23-33
RB trees
 allocating 21-3
 creating 21-3
 deleting 21-5
 initializing tree header data structure 21-3
 nodes
 adding to free list 21-5
 applying function to 21-4

busy, marking as 21-4
 collecting free nodes 21-5
 deleting 21-4
 determining number not busy 21-4
 determining whether deleted 21-4
 inserting into tree 21-3
 number of, determining 21-4
 placing on free list 21-4
 printing 21-4
 protection state 21-4
 searching for 21-3
 non-32 bits 21-5
 overview 21-2

RBFreeNodeCount function 21-4
 RBPrintTreeNode function 21-4
 RBRleasedNodeCount function 21-4
 RBTreeAddToFreeList function 21-5
 RBTreeBestNode function 21-3
 RBTreeCreate function 21-3
 RBTreeDelete function 21-4
 RBTreeFirstNode function 21-3
 RBTreeForEachNodeTilFalse function 21-4
 RBTreeGetFreeNode function 21-3
 RBTreeInsert function 21-3
 RBTreeIntInsert function 21-3
 RBTreeIntNearBestNode function 21-3
 RBTreeIntSearch function 21-3
 RBTreeLexiNode function 21-3
 RBTreeNearBestNode function 21-3
 RBTreeNextNode function 21-3
 RBTreeNodeDeleted function 21-4
 RBTreeNodeProtect function 21-4
 RBTreeNodeProtected function 21-4
 RBTreeNon32Bit function 21-5
 RBTreePrint function 21-4
 RBTreeSearch function 21-3
 RBTreeTrimFreeList function 21-5
 rc4_fill() function 3-61
 rcsmerge command E-22
 ready-to-run process 3-17
 Receive Trace Facility
 Latency Tracer API Functions 20-52
 Latency Tracer CLI Commands 20-53
 Latency Tracer Implementation 20-54
 periodic function 20-49
 snapshot display 20-52
 Storing Packet Information 20-46
 Time Threshold Messages 20-51
 Red-Black trees
 See RB trees
 redirect show command output 20-1
 Redundancy Facility
 See RF
 reference count
 packet buffer 5-26, 5-27

.reg file
 definition 14-13
 example 14-15
 format 14-14
 reg_add_entityapi_lookup_physical_entity_ext_confreg
 function 29-40
 reg_add_entityapi_lookup_physical_entity_ext_processor
 function 29-39
 reg_add_hc_hwcounter_get function 29-42
 reg_add_ifmib_get_operstatus_hwidb function 29-42
 reg_add_ifmib_hwifSpecific_get function 29-42
 reg_add_subsys_init_class function 7-25
 reg_invoke_entityapi_add_alias function 29-37
 reg_invoke_entityapi_add_Ipmap function 29-37
 reg_invoke_entityapi_add_logical_entity_struct
 function 29-37
 reg_invoke_entityapi_add_physical_entity
 function 29-37
 reg_invoke_entityapi_add_physical_entity_extension
 function 29-39
 reg_invoke_entityapi_delete_alias function 29-38
 reg_invoke_entityapi_delete_Ipmap function 29-38
 reg_invoke_entityapi_delete_logical_entity
 function 29-38
 reg_invoke_entityapi_delete_physical_entity
 function 29-37
 reg_invoke_entityapi_delete_physical_entity_extension
 function 29-39
 reg_invoke_entityapi_idbtype_to_vendoroid
 function 29-39
 reg_invoke_entityapi_IpTransportInfoToString
 function 29-39
 reg_invoke_entityapi_lookup_alias function 29-38
 reg_invoke_entityapi_lookup_Ipmap function 29-38
 reg_invoke_entityapi_lookup_logical_entity
 function 29-38
 reg_invoke_entityapi_lookup_physical_entity
 function 29-38
 reg_invoke_entityapi_test_physical_entity
 function 29-39
 reg_invoke_idb_proto_counter_get function 6-10
 reg_invoke_idb_proto_counter_increment function 6-11
 reg_invoke_idb_proto_counter_set function 6-11
 .regc file 14-13
 .regh file 14-13
 region manager
 definition 4-4
 memory hierarchy, defining 4-7
 region hierarchy, defining 4-10
 registering a region with 4-8
 registering region with (example) 4-8
 region_add_alias function 4-11
 (example) 4-11
 region_add_subalias function 4-11
 REGION_CLASS_FAST flag 4-9

REGION_CLASS_FLASH flag 4-9
REGION_CLASS_IMAGEBSS flag 4-9
REGION_CLASS_IMAGEDATA flag 4-9
REGION_CLASS_IMAGETEXT flag 4-9
REGION_CLASS_IOMEM flag 4-9
REGION_CLASS_LOCAL flag 4-9
REGION_CLASS_PCIMEM flag 4-9
region_create function 4-8
 (example) 4-8
region_exists function 4-13
region_find_by_addr function 4-12
 (example) 4-13
region_find_by_attributes function 4-12
region_find_by_class function 4-12
 (example) 4-13
region_find_next_by_attributes function 4-12
region_find_next_by_class function 4-12
 (example) 4-13
REGION_FLAGS_DEFAULT flag 4-12
REGION_FLAGS_INHERIT_CLASS flag 4-12
REGION_FLAGS_INHERIT_MEDIA flag 4-12
region_get_class function 4-14
region_get_media function 4-14
region_get_size_by_attributes function 4-13
region_get_size_by_class function 4-13
 example 4-14
region_get_status function 4-14
region_init function 4-40
REGION_MEDIA_ANY flag 4-10
REGION_MEDIA_READONLY flag 4-10
REGION_MEDIA_READWRITE flag 4-10
REGION_MEDIA_UNKNOWN flag 4-10
REGION_MEDIA_WRITEONLY flag 4-10
region_set_class function 4-8
region_set_media function 4-9
 (example) 4-10
REGION_STATUS_ALIAS flag 4-10
REGION_STATUS_ANY flag 4-10
REGION_STATUS_CHILD flag 4-10
REGION_STATUS_PARENT flag 4-10
regionlist
 initializing
 figure 4-39
regions
 figure 4-6
 adding subalias 4-11
 aliases
 declaring 4-11
 declaring (example) 4-11
 definition 4-10
 overview 4-11
 allocating
 transient memory allocation 4-38
attributes
 overview 4-6
retrieving (table) 4-14
setting (table) 4-14
child, definition 4-10
classes
 (table) 4-9
 definition 4-6
 hierarchy 4-7
 REGION_CLASS_FAST flag 4-9
 REGION_CLASS_FLASH flag 4-9
 REGION_CLASS_IMAGEBSS flag 4-9
 REGION_CLASS_IMAGEDATA flag 4-9
 REGION_CLASS_IMAGETEXT flag 4-9
 REGION_CLASS_IOMEM flag 4-9
 REGION_CLASS_LOCAL flag 4-9
 REGION_CLASS_PCIMEM flag 4-9
 retrieving 4-14
 setting 4-8
 creating 4-8
 creating (example) 4-8
 declaring 4-7
 definition 4-4
 determining if region exists 4-13
 dynamic region manager 4-38
 hierarchy
 figure 4-11
 aliases 4-10, 4-11
 child 4-10
 establishing 4-7, 4-10
 parent 4-10
 REGION_STATUS_ALIAS flag 4-10
 REGION_STATUS_ANY flag 4-10
 REGION_STATUS_CHILD flag 4-10
 REGION_STATUS_PARENT flag 4-10
 types (table) 4-10
 inheritance attributes
 REGION_FLAGS_DEFAULT flag 4-12
 REGION_FLAGS_INHERIT_CLASS flag 4-12
 REGION_FLAGS_INHERIT_MEDIA flag 4-12
 setting 4-12
 table 4-12
 media access attributes
 (example) 4-10
 REGION_MEDIA_ANY flag 4-10
 REGION_MEDIA_READONLY flag 4-10
 REGION_MEDIA_READWRITE flag 4-10
 REGION_MEDIA_UNKNOWN flag 4-10
 REGION_MEDIA_WRITEONLY flag 4-10
 retrieving 4-14
 setting 4-9
 table 4-9
 overview 4-6
 parent, definition 4-10
 parent-child hierarchy, determining 4-10
 region manager 4-4
 searching through 4-12

size
 (example) 4-14
 determining 4-13
status, determining 4-14
register declarations, using A-38
Register Storage Class A-20
register storage class, using A-20
Registries A-9
registries 14-13
 .c file
 definition 14-13
 .h file
 definition 14-13
 .reg file
 definition 14-13
 example 14-15
 format 14-14
 .regc file 14-13
 .rehg file 14-13
 adding names 14-56
 checking if specified tag is used 14-58
 common uses of 14-6
 compilation process 14-14
 default functions
 description 14-7
 definition 14-4
 deleting all callbacks 14-57
 deleting all names 14-57
 deleting callbacks 14-57
 deleting names from default list 14-57
 designing A-9
 files created by registry compiler 14-13
 invoking names 14-58
 libregistry.a 14-9
 metalanguage 14-14
 registry compiler, definition 14-14
 service initialization routines 14-13
 services, defining 14-13
 system registries 14-9
 wrapper functions 14-13
registries.mk 14-20
registry compiler, definition 14-14
registry functions
 true statements 27-32
Registry ReDefine Project 14-19
Registry services
 FASTCASE 14-38
registry services
 CASE 14-32
 CASE_LIST/CASE_LOOP 14-54
 FASTCASE 14-38
 FASTSTUB 14-47
 ILIST 14-27
 LIST 14-25
 LOOP 14-40
PID_LIST 14-30
RETVAL 14-36
SEQ_LIST 14-54
STUB 14-42
STUB_CHK 14-45
VALUE 14-52
related reading
 CNS 12-7
release note enclosures
 writing 35-1
reload command 2-42
REMOTE registry 14-8
remqueue function 22-4
reparenting E-10
req: property 13-5, 13-7
RES_BOOTP 27-5
RES_CONFIG 27-5
RES_DHCP 27-5
RES_HEURISTIC 27-5
RES_IPCP 27-5
RES_MANUAL 27-5
RES_NONVOL 27-5
RES_NULL 27-6
RES_PRIVATE_NONVOL 27-6
RES_RARP 27-6
RES_WRITING_NONVOL 27-6
reset_interrupt_level function 2-24, A-37
resolvemethod flags 27-5
resource policy command 20-23
restore_stdout function 20-66
retbuffer function 5-27, 5-48
retparticle function 5-46
return statement, spaces with A-25
RETVAL Services
 manipulating 14-36
retval services
 default function 14-38
retval services, description 14-3, 14-36, G-24
review
 submitting error messages for 19-22
RF
 APIs 24-44
 CLI support
 clear command 24-50
 configure command 24-50
 debug commands 24-49
 exec level commands 24-48
 show commands 24-48
 clients
 client IDs, rf_client_id_list.h 24-44
 description 24-30
 include files 24-43
 message call-back API 24-42
 message considerations 24-43
 message headers 24-44

message transport 24-31
messaging 24-41
header files 24-43
history
 CLI 24-46
 description 24-45
log function calls 24-45
log implementation details 24-45
log output, from the Active 24-46
log timestamps 24-47
standby logs (after a switchover) 24-47
syslog timestamps 24-47
infrastructure
 description 24-32
 fault management 24-33
 redundancy control and monitoring 24-32
 switchover notification input 24-33
MIB 24-58
platform code
 include files for supporting 24-43
platform-dependent support routines 24-50
 buffer routine registration 24-52
 ok to split 24-55
 ok to split registration 24-56
 open peer communication registration 24-55
 peer messaging 24-52
 peer messaging registration 24-52
 platform buffer management 24-50
 primary unit, designating 24-58
process configuration 24-57
reload command 24-57
reload registration 24-58
standby initialization 24-56
swact command processing 24-55
swact command registration 24-55
redundancy states
 active 24-36
 active progression 24-38
 active-drain 24-36
 active-fast 24-35
 active-postconfig 24-36
 active-preconfig 24-36
 description 24-33
 initialization 24-34
 initialization progression 24-38
 progression event call-back API 24-40
 progression events 24-36
 progression phases 24-37
 progression synchronization 24-39
 standby progression 24-39
 standby-bulk 24-35
 standby-cold 24-35
 standby-config 24-35
 standby-file system 24-35
 standby-hot 24-35
standby-negotiation 24-34
redundancy status events
 description 24-40
 status event call-back API 24-41
SNMP 24-58
sync buffer management 24-42
sync messages
 sending 24-42
RFC 1902 28-15
RFC 1903 28-15
RFC 1904 28-15
RFC 2233 1-11
RFCs
 854 28-9
 1213 28-32
 1516 28-8
 1573 28-6
 1902 28-15, 28-20
 1903 28-15
 1904 28-15
 1905 28-12
 2233 1573 28-20
 2573 29-43
 2578 28-15, 28-44
 2578 1212 28-7
 2578 1902 28-7
 2579 28-15
 2579 1903 28-8, 28-9
 2580 28-4, 28-14, 28-15
rings
 receive 23-9
 transmit 23-11
rn_addroute function 21-17
rn_delete function 21-18
rn_inithead function 21-17
rn_lookup function 21-18
rn_mark_parents function 21-18
rn_match function 21-18
rn_walktree function 21-17
rn_walktree_blocking function 21-17
rn_walktree_blocking_list function 21-18
rn_walktree_timed function 21-18
rn_walktree_version function 21-18
RNG 3-59
Rollback Component Conformance 27-52
ROM monitor
 bootstrapping Cisco IOS image 2-2
 calling an entry point to Cisco IOS image 2-17
 initializing a platform 2-2
Route Processor
 description 23-13
Route Switch Processor
 See RSP 23-14
routers
 essential components of 23-6

HA
 requirements 24-6

RP
 SSO
 switchover conditions 24-11
 switchover behavior
 SSO system platform arch. 24-10

RPCs, IPC
 setting timeout period 8-38
 simulating asynchronous response 8-38
 simulating send 8-38
 simulating synchronous response 8-38

RPF
 configuring 23-34
 description 23-34

RPR+
 defined 24-5
 description 24-8

RSP
 architecture overview 23-14

running process 3-17

runtime environments
 IOS code A-43

Rx Trace Server and Client API's 20-61
rx_trace_client_req function 20-52
rx_trace_coalesce function 20-52
rx_trace_coid_manager function 20-52
rx_trace_init_config function 20-52
rx_trace_init_parser function 20-52
rx_trace_periodic function 20-53
rx_trace_post function 20-53
rx_trace_server_req function 20-53
rx_trace_start function 20-53
rx_trace_stop1 function 20-53
rx_trace_stop2 function 20-53

S

s_tohigh function D-12
 See process_create function

s_tolow function D-13
 See process_create function

safe library functions 11-7

SAL 23-81

Sarbanes-Oxley 27-45, 27-58, 27-63

SB_DEFAULT_ACCESS define constant 6-28

SB_LIST_ACCESS define constant 6-28

scalability 32-1

scalers@cisco.com 32-1

scatter-gather DMA
 See particles

SCHED_THRASH_THRESHOLD 3-54

scheduler
 bit fields

 changing minor identifier 3-44
 clearing specified bits 3-45
 creating 3-44
 definition 3-44
 deleting 3-45
 determining reason process awoken is a changed bit field 3-45
 registering a process on 3-44, 3-49
 retrieving value of 3-45
 setting specified bits 3-45
 specifying only one process awoken 3-45

booleans
 changing minor identifier 3-43
 creating 3-43
 definition 3-43

commands
 heapcheck poll 3-15
 heapcheck process 3-15
 max-task-time 3-16, 3-29
 process-max-time 3-15
 process-watchdog 3-16, 3-29
 run-degraded 3-16
 scheduler isr-watchdog 3-30
 commands, important 3-15
 compatibility queues D-2
 example 3-28, 3-38
 heapcheck poll command 3-15
 heapcheck process command 3-15
 housekeeping operations D-3
 idle queue 3-8
 manage processes
 description 3-18
 managed object tasks 3-39
 managed objects, types 3-39
 max-task-time command 3-16, 3-29
 messages 3-49
 determining if queued 3-49
 determining if simple message is queued 3-49
 registering a process for notification 3-49
 retrieving 3-49
 sending at interrupt level 3-48
 sending to 3-49
 new, definition D-1
 non-preemptive 3-6
 old, definition D-1
 overview 3-6
 process management
 description 3-18
 process states
 dead 3-18
 hung 3-18
 ready to run 3-17
 running 3-17
 sleeping (absolute time) 3-17
 sleeping (interval) 3-17

sleeping (managed timer) 3-18
sleeping (periodic) 3-18
table 3-17
waiting for event 3-17
processes
 description 3-16
 moving between queues 3-9
 PID 3-20
 priority 3-16
 stopping 3-27
 process-max-time command 3-15
 process-watchdog command 3-16, 3-29
queues
 adding data pointer to 3-42
 changing minor identifier 3-41
 critical, operation D-3
 critical-priority, operation (figure) D-4
 definition 3-41
 high-priority, operation D-4
 high-priority, operation (figure) D-5
 locating first item on 3-42
 low-priority, operation D-5
 low-priority, operation (figure) D-7
 medium-priority, operation D-5
 medium-priority, operation (figure) D-7
operation, description 3-9, D-2
operation, description (figure) D-3
priority (figure) D-2
registering a process on 3-41
specifying only one process awoken per
 enqueue 3-41
types of 3-7
run-degraded command 3-16
selection algorithm 3-9
semaphores
 changing minor identifier 3-46
 creating 3-46
 definition 3-39
 deleting 3-47
 locking 3-46
 locking atomically 3-47
 managed 3-46
 ordering lock based on time 3-47
 simple 3-47
 unlocking 3-46
 unlocking atomically 3-47
 watched 3-46
terminology 3-2
threads
 See scheduler, processes
timer expiration 3-49
types of managed queues 3-41
scheduler isr-watchdog command 3-30
SCTP
 functions 10-3
one-to-many
 (example) 10-8
one-to-one
 (example) 10-4
sdb_add_string function 21-19
sdb_find_string function 21-20
sdb_register_component function 21-19
sdb_remove_string function 21-20
sdb_string_addr function 21-20
seat manager, IPC, definition 8-4
seat table, IPC
 description 8-23
 entries 8-23
seats, IPC
 adding 8-23
 definition 8-4
 resetting 8-24
 retrieving from seat table 8-23
secs_and_nsecs_since_jan_1_1970 function 15-4
secure mode
 password validation 11-12
secure-ios@cisco.com 30-1
security
 coding issues A-42
 infinite loops and MAX value usage 30-13
 key storage issues (example) 30-12
 printf() 30-7
 printf(), sprintf(), and buginf() format strings 30-7
 puts() 30-8
 strcpy(), strcat(), and sprintf() issues 30-9
 strings 30-2
 vulnerability 4-86
 zeroing passwords and keys 30-11
Security Initiative
 overview 11-2
Segment and Session Information Handling Details 23-80
Segment Switching Manager 23-80, 23-82
Segment-switching Abstraction Layer 23-81
semaphores
 changing minor identifier 3-46
 creating 3-46
 definition 3-39
 deleting 3-47
 locking 3-46
 locking atomically 3-47
 managed
 definition 3-46
 ordering lock based on time 3-47
 simple
 definition 3-47
 locking 3-47
 unlocking 3-47
 unlocking 3-46
 unlocking atomically 3-47
 watched, definition 3-46

SEP
See ESM

seq: property 13-5, 13-6

SEQ_LIST Services
manipulating 14-58

sequencing
link order 13-6
subsystems 2-20

service config command 7-40

service point, definition 14-23

Service Selection Gateway 23-1

services
case services
adding (example) 14-34, 14-39
adding default (example) 14-36
default function 14-35
defining 14-33, 14-38
defining (example) 14-33, 14-39
description 14-3, 14-32, 14-38, G-4
invoking (example) 14-35, 14-40
wrapper functions 14-33, 14-38
definition 14-4, 14-23

fastcase services
default function 14-40

faststub services
default function 14-51

ilist services
default function 14-30

list services
adding (example) 14-26, 14-28
default function 14-27
defining 14-25, 14-27
(example) 14-25
defining (example) 14-28
description 14-3, 14-27, G-13
invoking (example) 14-26, 14-30
wrapper functions 14-25, 14-27

loop services
adding (example) 14-41
default function 14-42
defining 14-40
defining (example) 14-41
description 14-3, 14-40, G-13
invoking (example) 14-42
wrapper functions 14-40

pid_list services
adding (example) 14-32
default function 14-32
defining 14-31
defining (example) 14-31
description 14-3, 14-30, G-19
invoking (example) 14-32
wrapper functions 14-31

retval services
default function 14-38

retval services, description 14-3, 14-36, G-24

service point, definition 14-23

stub services
adding (example) 14-44, 14-46, 14-50
default function 14-44
defining 14-43, 14-45, 14-48
defining (example) 14-43, 14-46, 14-48, 14-49, 14-50
description 14-3, 14-42, 14-45, G-28
invoking (example) 14-44, 14-47, 14-51
wrapper functions 14-43, 14-45, 14-48

stub_chk services
default function 14-47

types of 14-2

value services
adding (example) 14-53
adding default (example) 14-54
defining 14-52
defining (example) 14-52
description 14-4, 14-52, G-30
invoking (example) 14-53
wrapper functions 14-52

set memory debug incremental starting-time
command 4-44

set_crashinfo_bufsize function 7-19

set_if_input function 5-37

set_interrupt_level function 2-24

setting up #include's and #define's
error messages 19-7

severity
error message values (table) 19-9

sh mem 0x 4-42

Shared Information Utility 4-89
dataset registration 4-91
group registration 4-91

shared memory
architecture 23-41

shminfo_data_ver_update function 4-93, 4-95

shminfo_dataset_deregister function 4-91

shminfo_dataset_register function 4-91

shminfo_get_is_ready function 4-91

shminfo_get_my_data function 4-92

shminfo_group_deregister function 4-92

shminfo_group_register function 4-91

shminfo_reader_lock_version function 4-93

shminfo_reader_unlock_version function 4-93

shminfo_registry.h file 4-95

shminfo_writer_commit_cb_add function 4-94

shminfo_writer_data_abort function 4-94

shminfo_writer_data_committable function 4-94

shminfo_writer_data_get_writeptr function 4-93

shminfo_writer_disable_commit function 4-94

shminfo_writer_enable_commit function 4-94

shortcuts 27-2

show

process memory 23-15
show archive differences command 27-90
show buffers command 23-8
show buffers command output 5-53
show buffers enabled commands 5-53
show cef interface internal command 23-59, 23-67
show chunk command 4-33, 4-44
show controller cbus command 23-15
show if-mgr db interface command 29-22
show ip cef internal command 23-52
show ipc nodes command 8-54
show ipc port command 8-54
show ipc queue command 8-53
show ipc rpc command 8-55
show ipc session command 8-54
show ipc session rx verbose 8-51, 8-57
show ipc session tx verbose 8-51, 8-55
show ipc status 8-51
show ipc status cumulative command 8-53
show ipc zones command 8-55
show mem dead command 4-44
show memory allocating-process command 4-44
show memory command 4-44
 output, transient memory 4-42
show memory dead command 4-44
show memory debug incremental command 4-44
show memory debug leaks command 4-44, 20-23
show memory debug references command 4-44
show memory ecc command 4-44
show memory failures alloc command 4-44
show memory failures allocation command 4-22
show memory fast command 4-44
show memory fragment command 4-44
show memory free command 4-44
show memory io command 5-53
show memory multibus command 4-44
show memory pci command 4-44
show memory processor command 4-44
show memory scan command 4-44
show memory statistics history table command 4-43
show parser links command 27-71
show proc cpu command 4-57
show proc mem command 4-49
show process event pid 3-58
show processes all-events 3-59
show processes memory command 4-45
show profile command C-5
show region command 4-45
show region command output
 transient memory 4-41
show run 27-73
show running config command 27-98
show running-config command 27-98
show stack pid 3-59
show tech-support cef command 23-52

show version command 4-45
show warm-reboot command 2-42
signal_oneshot function 18-2
 (example) 18-3
signal_permanent function 3-28, 18-3
 (example) 18-3
signal_send function 18-4
 (example) 18-4
signals
 exception 18-1
 sending 18-4
signed types, in code A-21
Silicon Switching
 description 23-12
Silicon Switching Engine
 See SSE 23-13
simple file system 9-3
Simple Network Management Protocol
 See SNMP
singly linked lists
 types of 22-1
 with queuing blocks, See indirect queues
 See also queues, direct; queues, indirect
SLEEPING macro 16-5
 guidelines for using 16-5
sleeping process 3-17, 3-18
SMI
 ASN.1 application types 28-8
 ASN.1 primitive data types 28-8
 components 28-2
 definition 28-2
 overview 28-8
 textual conventions 28-9
smicng file 28-21
SNMP
 agent, definition 28-2
 applications, design considerations 28-11
 asynchronous notifications
 controlling 28-36
 defining 28-35
 description 28-3
 generating 28-38
 implementing 28-35
 informs, definition 28-3
 location 28-35
 snmp-server enable command 28-36
 snmp-server host command 28-36
 traps, definition 28-3
 conceptual tables
 complex 28-7
 definition 28-6
 index objects, coding 28-7
 simple 28-6
 tables inside tables 28-7
 manager, definition 28-2

- modularity, observing 28-33
- notification 29-42
- operations 28-42
- overview 28-1 to 28-3
- RFCs 28-15
- security facilities 28-3
- textual conventions 28-9
- transport protocols 28-3
- SNMP Infra Tool 28-22
- SNMP Management Information Compiler Next Generation 28-21
 - snmp mib persist command 29-40
 - snmp-server host command 28-36
 - snmp-servier enable command 28-36
 - snmp-servier host command 28-36
 - snprintf function 17-2
 - socket structure,setting 3-26
 - sockets
 - functions and macros
 - (table) 10-1
 - SCTP 10-3
 - software-d@cisco.com A-1, B-1
 - software-questions@cisco.com A-1
 - soutput vector 6-39
 - Spaces around Operators A-25
 - Spaces around Parentheses A-25
 - sprintf function 17-2, A-37
 - security issues 30-9
 - SRAM (MEMD)
 - RSP 23-14
 - SSE
 - description 23-13
 - SSM 23-80
 - SSO
 - Checkpointing Facility (CF)
 - description 24-25
 - hardware
 - architectural assumptions 24-8
 - automatic switchover, a fault on the active RP 24-11
 - automatic switchover, active RP declared "dead" 24-11
 - automatic switchover, criteria-based 24-12
 - line card behavior 24-10
 - manual switchover, CLI invoked 24-11
 - RP switchover behavior 24-10
 - RP switchover conditions 24-11
 - system model 24-8
 - system platform architecture 24-8
 - trial switchover, CLI invoked 24-11
 - implementation
 - overview 24-17
 - IOS infrastructure components
 - boot image 24-23
 - configuration issues 24-23
 - configuration register 24-23
 - Configuration Synchronization (Config Sync) 24-22
 - description 24-22
 - file system 24-23
 - line card images 24-23
 - OS-logging 24-24
 - parser return codes 24-23
 - private configuration 24-24
 - remote file systems 24-24
 - rommon variables 24-23
 - SNMP information 24-24
 - startup configuration 24-23
 - system memory 24-24
 - IPC
 - description 24-25
 - Fragmentation and Reassembly (FAR) 24-26
 - protocol versioning 24-25
 - windowing 24-25
 - MIB
 - variables 24-28
 - network management support 24-28
 - platform specific support
 - APS support 24-28
 - ATM drivers and line cards 24-28
 - description 24-27
 - IPC 24-28
 - line card drivers 24-27
 - post switchover state verification and reconciliation 24-28
 - RF 24-27
 - platforms
 - initial target set 24-17
 - overview 24-58
 - Redundancy Facility (RF)
 - description 24-24
 - routed protocols
 - Asynchronous Transfer Mode (ATM) 24-21
 - description 24-20
 - Frame Relay (FR) 24-21
 - Point to Point Protocol (PPP) 24-20
 - routing protocols
 - Border Gateway Protocol (BGP) 24-19
 - Intermediate System-to-Intermediate System Interior Gateway routing protocol (IS-IS) 24-20
 - Open Shortest Path First protocol (OSPF) 24-19
 - SNMP data 24-28
 - software
 - architectural model, description 24-12
 - architectural model, driver-client 24-13
 - architectural model, model 1 24-13
 - architectural model, model 1 vs. model 3 24-15
 - architectural model, model 3 24-15
 - Cisco express forwarding (CEF) 24-18
 - coredump 24-26
 - event tracer 24-27
 - fast software upgrade (FSU) 24-17

features 24-12, 24-17
graceful shutdown 24-20
hitless software upgrade (HSU) 24-17
media layer (a.k.a. network.c) 24-26
non ha-aware protocols and features 24-16
routing protocols and non-stop forwarding
(NSF) 24-17
upgrade, proposed methods 24-16
target platform architectures 24-58
SSS 23-82
stack corruption
troubleshooting 20-74
stakeholders in branching E-26
Standard Cisco Header A-22
Standard Indentation
rules & exceptions A-22
utility for checking diffs A-23
standard libraries
ANSI C functions 1-ix, 11-16
POSIX functions 1-ix, 11-16
standby RP
defined 24-5
stateful switchover
See SSO
Static Analysis 37-2
Static Class A-20
static memory
allocating 4-38
interleaving with transient memory allocations 4-38
static pools 5-7
static storage class, using A-20, A-27
static_ios command 19-19
storage
backing store (output queue) 23-17
non-volatile 7-22
storage classes, using A-27
store_persistence_data function 29-40
strcasecmp_s function 11-9
strcasecmp_s function 11-9
strcat function
security issues 30-9
strcat_s function 11-10
strcmp_s function 11-10
strcmpfld_s function 11-10
strcpy function
security issues 30-9
strcpy_s function 11-10
strcpyfld_s function 11-10
strcpyfldin_s function 11-10
strcpyfldout_s function 11-10
strcspn_s function 11-11
Stream Control Transmission Protocol
See **SCTP**
strfirstchar_s function 11-11
strfirstdiff_s function 11-11
strfirstsame_s function 11-11
string database
adding strings 21-19
API overview 21-19
description 21-19
finding strings 21-20
registering 21-19
removing strings 21-20
string address 21-20
strings 11-12
copying 11-10
formatting
AppleTalk addresses 17-8
Banyan VINES addresses 17-10
placing into buffer 17-2
timestamps 17-5
handling run-time constraints 11-8
left justifying 11-13
secure code
zeroing 11-15
secure mode
checking alphanumeric characters 11-11
comparing 11-10
computing length 11-13
computing prefix length 11-14
concatenating 11-10, 11-13
converting from lowercase to uppercase 11-15
converting from uppercase to lowercase 11-15
copying 11-13
copying to arrays 11-10
counting characters not in string 11-11
determining prefix 11-14
first occurrence of substring 11-14
removing whitespaces 11-14
retrieving first occurrence of character 11-11
retrieving index of differing character 11-11
retrieving index of same character 11-11
retrieving index to last differing character 11-13
retrieving index to last same character 11-13
retrieving next token 11-14
retrieving pointer to character contained in both
strings 11-14
retrieving pointer to last occurrence of
character 11-12
whether containing ASCII 11-11
whether containing digits 11-12
whether containing hex 11-12
whether lowercase 11-12
whether mixed case 11-12
whether uppercase 11-12
security mistakes 30-2
unicode 20-65
strings, ANSI C 17-1
strings, formatting
debugging messages 17-1

time 17-4
user command output 17-1
`strisalphanumeric_s` function 11-11
`strisascii_s` function 11-11
`strisdigit_s` function 11-12
`strishex_s` function 11-12
`strislowercase_s` function 11-12
`strismixedcase_s` function 11-12
`strispASSWORD_s` function 11-12
`strisUPPERCASE_s` function 11-12
`strlastchar_s` function 11-12
`strlastdiff_s` function 11-13
`strlastsame_s` function 11-13
`strljustify_s` function 11-13
`strncat_s` function 11-13
`strncpy_s` function 11-13
`strnlen_s` function 11-13
`strpbrk_s` function 11-14
`strprefix_s` function 11-14
`strremovews_s` function 11-14
`strspn_s` function 11-14
`strstr_s` function 11-14
`strtok_s` function 11-14
`strtolowercase_s` function 11-15
`strtouppercase_s` function 11-15
Structure of Management Information
 See SMI
structures
 parser transition structure 27-6
 `rx_trace_buffer_` 20-47
 `rx_trace_efifo_` 20-48
 `rx_trace_entry_` 20-47
 `strzero_s` function 11-15
stub functions, not using A-9
STUB Services
 manipulating 14-42
stub services
 adding
 (example) 14-44, 14-46, 14-50
 default function 14-44
 defining 14-43, 14-45, 14-48
 (example) 14-43, 14-46, 14-48, 14-49, 14-50
 description 14-3, 14-42, 14-45, G-28
 invoking
 (example) 14-44, 14-47, 14-51
 wrapper functions 14-43, 14-45, 14-48
STUB_CHK Services
 manipulating 14-45
stub_chk services
 default function 14-47
Stubbing Out code A-25
stubbing out code A-25
style
 VM coding and mathematical notations 4-79
subblocks

 adding to IDB 6-21
 allocating ID constants 6-28
 deleting from IDB 6-22
 dynamic, description 6-21
FIB
 see FIB 23-55
obtaining pointer
 to hardware IDB 6-22
 to software IDB 6-22
subiabtype data structure 1-11, 29-31
subinterfaces
 freeing 6-6
 unlinking 6-5
submode
 alternate mode 27-75
 exit to parent mode 27-75
 parser 27-75
Subscriber Service Switch 23-1, 23-82
`SUBSYS_CLASS_DRIVER` 13-4
`SUBSYS_CLASS_DRIVER` flag 13-4
`SUBSYS_CLASS_EHSA` flag 13-4, 13-6
`SUBSYS_CLASS_IFS` flag 13-4, 13-6
`SUBSYS_CLASS_KERNEL` flag 13-3, 13-6, 13-7
`SUBSYS_CLASS_LIBRARY` flag 13-4, 13-7
`SUBSYS_CLASS_MANAGEMENT` flag 13-4, 13-7
`SUBSYS_CLASS_PRE_DRIVER` flag 13-4, 13-6
`SUBSYS_CLASS_PRE_EHSA` flag 13-4, 13-6
`SUBSYS_CLASS_PROTOCOL` flag 13-4, 13-7
`SUBSYS_CLASS_REGISTRY` flag 13-3, 13-6
`SUBSYS_CLASS_SYSINIT` flag 13-3, 13-6
`SUBSYS_CLASS_UCODE` flag 13-4
`SUBSYS_HEADER` macro 13-8, 13-9
subsystems
 classes
 choosing 13-5
 creating 13-10
 defining 13-8
 (example) 13-9
 description B-4
 designing A-8
 entry point 13-2
 header
 defining (example) 13-9
 header, defining 13-8
 init function 13-2
 initializing 13-2
 properties 13-5
 req: property 13-5, 13-7
 requirements property 13-5, 13-7
 runtime 13-2
 seq: property 13-5, 13-6
 sequencing 2-20
 sequencing property 13-5, 13-6
 structure, filling 13-9
 subsystype structure 13-9

tips for creating 13-10
substype structure 13-9
summer time
 description 15-3
 testing for 15-4
Support for Layer 2 to Layer 2 Point-to-Point
 Connections 23-74
Support for Locally-Terminated Tunnels 23-72
Switch Processor
 description 23-13
Switch Statements and Default Cases A-28
switching
 "routing" 23-11
 "slow switching" 23-11
 10000 23-27
 7000-specific systems 23-12
 720x 23-24
 75xx, 720x/71xx series 23-13
 Autonomous 23-13
 CEF
 adding a CEF feature, FIB IDBs 23-44
 adding a CEF feature, overview 23-44
 description 23-20
 example forwarding code 23-48
 FIB subblocks, see 23-55
 FIB, overview 23-20
 load balancing 23-51
 resources 23-54
 disadvantages of demand-based caching 23-19
 distributed
 compatibility matrix 23-30
 FIB subblocks 23-63
 VIP 23-22
 fast switching
 overview 23-11
 with a non-VIP LC 23-15
 writing code 23-35
 GSR 23-13
 high-end systems 23-13
 load balancing along paths 23-32
 low- and mid-range systems 23-8
NetFlow
 description 23-18
 performance 23-19
on non-VIP-based LC 23-15
Optimum
 description 23-17
 non-VIP LC 23-15
paths
 types 23-7
process switching
 Input, Background, Router (3 processes) 23-11
 overview 23-10
 with a non-VIP LC 23-16
RSP CEF
 (figure) 23-21
Silicon 23-13
VIP
 dCEF and distributed switching 23-22
switchover
 defined 24-5
RP
 SSO system platform arch. 24-10
 RP conditions for SSO 24-11
SWSB_FAST define constant 6-28
sync
 incremental E-4
syncing E-20
 advantages and disadvantages E-23
 types of E-3
sys_timestamp structure 16-3
syslog-dev@cisco.com 20-1
system availability
 HA
 definition 24-5
system clock 15-2
 changes 15-3
 description 15-2
 setting 15-5
system initialization
 basic 2-2
 by ROM monitor 2-2
 description 2-1 to 2-20
 fundamental (figure) 2-18
 of Cisco IOS image 2-19
system registries
 summary 14-9
system startup
 parsing 24-22
system_uptime_seconds function 16-34

T

table
 adjacency (CEF)
 populating 23-20
tables
 adjacency (CEF)
 description 23-20
TAC engineer
 identifying error message information for 19-18
tag
 flow (NetFlow Switching) 23-18
TCL 20-65
temporary pool items 5-8
terminal lines
 initialization 7-32
terminal prc expose 27-40
terminal prc hide 27-40

TEST_MULTIPLE_FUNCS macro 27-19
testing
 error messages 19-19
TEXT areas
 virtual memory 4-76
The Receive Trace Facility Overview 20-46
THIS_PROCESS flag 3-27
Thrashing 3-54
threads
 See processes
threshold messages 20-51
throttle_netio function 2-21
throttling
 pool cache handling 5-17
time
 determining current 16-9
time formats
 clock_epoch structure 15-2
 convert between 15-5
 timeval structure 15-2
 UNIX format 15-2
time of day
 clock/calendar, in hardware 15-3
 current time, getting 15-4
 daylight savings time
 description 15-3
 testing for 15-4
 epoch
 clock_epoch structure 15-1
 definition 15-1
 NTP 15-3
 summer time
 description 15-3
 testing for 15-4
system clock
 description 15-2
 setting 15-5
time formats
 clock_epoch structure 15-2
 convert between 15-5
 timeval structure 15-2
 UNIX format 15-2
time source, determining 15-4
time strings, formatting 15-5, 17-4
time validity, determining 15-5
time zone name 15-4
time zone offset, determining 15-4
time zones 15-3
time, format for printing 17-4
TIME_LEFT_SLEEPING macro 16-5
TIME_LEFT_SLEEPING64 macro 16-5
Timer Wheel Granularity 16-20
Timer Wheel Timers 16-19
Timer Wheel timers 16-1
Timer Wheel Timers Benefits 16-24

TIMER_ADD_DELTA macro 16-7
TIMER_ADD_DELTA64 macro 16-7
TIMER_EARLIER macro 16-10
TIMER_LATER macro 16-10
TIMER_RUNNING macro 16-5
TIMER_RUNNING_AND_AWAKE macro 16-5
TIMER_RUNNING_AND_SLEEPING macro 16-5
TIMER_SOONEST macro 16-6
TIMER_START macro
 (example) 16-7, 16-8
 prototype 16-4
 TIMER_START_ABSOLUTE macro 16-4
 TIMER_START_ABSOLUTE64 macro 16-4
 TIMER_START_GRANULAR macro 16-4
 TIMER_START_GRANULAR64 macro 16-4
 TIMER_START_JITTERED macro 16-4
 TIMER_START64 macro 16-4
 TIMER_STOP macro 16-5
 TIMER_SUB_DELTA macro 16-7
 TIMER_SUB_DELTA64 macro 16-7
 TIMER_UPDATE macro 16-6
 TIMER_UPDATE_GRANULAR macro 16-4
 TIMER_UPDATE_GRANULAR64 macro 16-4
 TIMER_UPDATE_JITTERED macro 16-6
 TIMER_UPDATE64 macro 16-6
timers
 context value 16-11
 jitter 16-2
 See managed timers, passive timers in the future, passive
 timers in the past
 system clock 16-2
 type value 16-11
TIMERS_EQUAL macro 16-6
TIMERS_NOT_EQUAL macro 16-6
timestamps
 comparing 16-6, 16-10
 copying 16-9
 copying atomically 16-9
 current, obtaining 16-9
 description 16-2
 earlier, determining 16-6
 elapsed time, determining 16-9
 equality
 determining whether equal 16-6
 determining whether unequal 16-6
 file systems 9-22
 formatting 17-5
 sys_timestamp structure 16-3
 system clock 16-2
 testing whether time is within bounds 16-9, 16-10
 See also passive timers in the past
timeval structure 15-2
 Cisco IOS definition 15-1
 POSIX definition 15-1
TL9K 27-45, 27-63

tm_malloc function 4-19, 4-40
tm_malloc_aligned function 4-19, 4-41
tokens, parsing
 keyword-number combinations 27-17
 keyword-number combinations (example) 27-17
 keywords 27-10
 keywords (example) 27-12
 numbers 27-13
 numbers (example) 27-16
 optional keywords 27-18
Tool Command Language 20-65
TRAILBYTES packet data padding 5-20
transient memory 4-17
 allocating 4-38
 (example) 4-38
 interleaving with static memory allocations 4-38
 separate regions 4-38
 API functions 4-40
 CHUNK_FLAGS_TRANSIENT flag 4-41
 CLI output 4-41
 dynamic region manager 4-38
 enabling 4-38
 freeing memory to add to freeregionlist 4-40
 initializing freeregionlist 4-38
 managing memory pools 4-40
 memory pools
 description 4-38
 show memory command output 4-42
 show region command output 4-41
transition structure, parser 27-6
transmit queue
 backing store 23-17
traps
 generating 29-43
 SNMP 29-42
traps, definition 28-3
trees, binary
 See AVL trees, radix trees, RB trees
tunneling
 fragmenting packet buffers 5-28
tw_process_expiry function 16-33
tw_timer_get_type function 16-27
tw_timer_get_type_ext function 16-27
tw_timer_granularity function 16-27
tw_timer_init_wheel function 16-27
tw_timer_init_wheel_ext function 16-27
tw_timer_load_tbl function 16-27
tw_timer_remaining function 16-27
tw_timer_running function 16-27
tw_timer_running() function 16-27
tw_timer_set_handler function 16-27
tw_timer_set_type function 16-27
tw_timer_set_type_ext function 16-27
tw_timer_start function 16-27
tw_timer_start_jittered function 16-27

tw_timer_stop function 16-27
tw_timer_tick function 16-27, 16-31
Tx Ring/Queue
 description 23-11
Typecasting A-20
typecasting
 exception A-20
typecasting, in code A-20

U

Unicast Reverse Path Forwarding
 See RPF 23-34
Unicode string functions 20-65
Unit Test 37-2
UNIX format 15-1
unix_time function 15-4
unix_time_is_in_summer function 15-4
unix_time_string function 15-6
unix_time_string_2 function 15-6
unix_time_to_epoch function 15-5
unix_time_to_timeval function 15-5
unprofile command C-4
unprofile task command C-5
unprovision 23-85
unqueue function 22-4
 (example) 22-5
unsafe and safe functions 11-3
unsigned types, in code A-21
unwedging an input queue 20-23
update-mibs.pl script, invoking 28-21
URLs, debugging-related documentation 20-27

V

VALUE Services
 manipulating 14-52
value services
 adding (example) 14-53
 adding default (example) 14-54
 defining 14-52
 (example) 14-52
 description 14-4, 14-52, G-30
 invoking (example) 14-53
 wrapper functions 14-52
values
 severity, error message (table) 19-9
VFI 23-98
VIP
 dCEF and distributed switching 23-22
 description 23-22
virtual address

- definition 4-82
 - discussion 4-75
 - virtual addresses vs. physical addresses 4-75
 - Virtual Ifc 23-81
 - Virtual Interface 23-81
 - virtual memory
 - addressing basics 4-75
 - advice on using 4-76
 - basic terms and concepts 4-80
 - benefits and costs 4-73
 - coding and mathematical notation style 4-79
 - definition 4-80
 - engineering effort to port 4-74
 - overview of Cisco IOS implementation 4-73
 - Paging Game, a humorous introduction 4-72
 - requirements 4-73
 - rules of Cisco IOS 4-75
 - steps in porting to a platform 4-77
 - TEXT and LOCAL 4-76
 - Virtual Private LAN Service 23-81
 - Virtual Private Network 23-1
 - VM
 - see virtual memory 4-72
 - volatile keyword, using A-38
 - VPLS 23-81
 - VPN
 - hiding internal networks
 - (example) 30-20
 - VRF 30-19
 - VRF-aware DNS 25-7
 - VRFifying 30-19
 - VRFs
 - private 30-19
 - W**
 - waiting-for-event process 3-17
 - wall clock time 15-2
 - Warm Reboot 2-42
 - memory requirements 2-58
 - Warm Reboot Storage 2-42
 - Warm Upgrade
 - API functions 2-59
 - CLI
 - description 2-59
 - modifications to existing CLIs 2-59
 - new 2-59
 - code
 - common 2-60
 - image format-specific 2-60
 - platform-specific 2-60
 - processor-specific 2-60
 - comparison to Warm Reboot 2-58
 - components 2-58
 - description 2-57
 - flow overview 2-59
 - warm-reboot command 2-42
 - watchdog
 - preemptive processes 3-31
 - process accounting 3-29
 - scheduler isr-watchdog command 3-30
 - watched bit field
 - See bit fields
 - watched boolean
 - See booleans
 - watched timers
 - changing 16-33
 - WAVL trees
 - See AVL trees
 - wavl_delete function 21-12
 - wavl_delete_thread function 21-12
 - wavl_finish function 21-12
 - wavl_get_first function 21-10
 - wavl_get_next function 21-10
 - wavl_init function 21-8
 - wavl_insert function 21-9
 - wavl_insert_thread function 21-9
 - wavl_normalize function 21-12
 - wavl_remove
 - prototype 21-7
 - wavl_remove function 21-7
 - wavl_search function 21-11, 21-12
 - wavl_walk function 21-10
 - Weighted Fair Queuing
 - backing store 23-17
 - words
 - extracting from byte stream 26-14
 - inserting into byte stream 26-14
 - working set
 - definition 4-82
 - wrapped AVL trees
 - See AVL trees, wrapped
 - wrb_saved_hdr_t data structure 2-54
 - write checkpoint command 27-98
 - write memory command 27-98
 - write-rnes@cisco.com 35-1
 - Writing Cisco IOS Code
 - Style Issues A-1
 - Writing Code for a Scalable Interface Type A-40
- X**
 - XAWAKE macro 16-5
 - guidelines for using 16-5
 - XSLEEPING macro 16-5
 - guidelines for using 16-5

Z

Zero NVGEN 27-61
zone manager, IPC
 definition 8-5
zones
 closing seats 8-60
 creating 8-59
 getting by name 8-59
 getting data 8-60
 getting name 8-60
 managing 8-58
 removing 8-60
 setting local seat address 8-61
 setting platform ack vector 8-60
 setting platform master control id 8-60
 setting platform seat id 8-60
 setting platform seat name 8-61
 setting platform transport 8-61
 setting platform transport type 8-61
 setting platform tx vector 8-61
 setting seat master 8-61
zones, IPC
 definition 8-4

CISCO CONFIDENTIAL