CCIE Professional Development

Inside Cisco IOS Software Architecture

An essential guide to understanding the internal operation of Cisco routers

Vijay Bollapragada, CCIE® No. 1606
Curtis Murphy, CCIE No. 1521
Russ White, CCIE No. 2635

ciscopress.com

Inside Cisco IOS Software Architecture

Vijay Bollapragada, Curtis Murphy, & Russ White

Copyright© 2000 Cisco Systems, Inc.

Cisco Press logo is a trademark of Cisco Systems, Inc.

Published by:
Cisco Press
800 East 96th Street
Indianapolis, IN 46240 USA

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the publisher, except for the inclusion of brief quotations in a review.

Printed in the United States of America 4 5 6 7 8 9 0

Fourth Printing May 2009

Library of Congress Cataloging-in-Publication Number: 99-64092

ISBN: 1-58705-816-2

Warning and Disclaimer

This book is designed to provide information about Cisco's proprietary Internetwork Operating System (IOS) software architecture. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an "as is" basis. The author, Cisco Press, and Cisco Systems, Inc., shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the author and are not necessarily those of Cisco Systems, Inc.

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Cisco Press or Cisco Systems, Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book, or otherwise alter it to better suit your needs, you can contact us through e-mail at ciscopress@mcp.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Publisher	John Wait
Editor-In-Chief	John Kane
Cisco Systems Program Manager	Jim LeValley
Managing Editor	Patrick Kanouse
Senior Acquisitions Editor	Brett Bartow
Development Editor	Christopher Cleveland
Project Editor	Jennifer Nuckles
Copy Editor	Theresa Wehrle
Technical Editors	Mike Brown, Jennifer DeHaven Carroll, Ron Long, Alexander Marhold
Team Coordinator	Amy Lewis
Book Designer	Gina Rexrode
Cover Designer	Louisa Adair
Production Team	Argosy
Indexer	Kevin Fulcher

CISCO™

Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV
Amsterdam, The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at **www.cisco.com/go/offices.**

CCDE, CCENT, Cisco Eos, Cisco Lumin, Cisco Nexus, Cisco StadiumVision, the Cisco logo, DCE, and Welcome to the Human Network are trademarks.; Changing the Way We Work, Live, Play, and Learn is a service mark; and Access Registrar, Aironet, AsyncOS, Bringing the Meeting To You, Catalyst, CCDA, CCDP, CCIE, CCIP, CCNA, CCNP, CCSP, CCVP, Cisco, the Cisco Certified Internetwork Expert logo, Cisco IOS, Cisco Press, Cisco Systems, Cisco Systems Capital, the Cisco Systems logo, Cisco Unity, Collaboration Without Limitation, EtherFast, EtherSwitch, Event Center, Fast Step, Follow Me Browsing, FormShare, GigaDrive, HomeLink, Internet Quotient, IOS, iPhone, iQ Expertise, the iQ logo, iQ Net Readiness Scorecard, iQuick Study, IronPort, the IronPort logo, LightStream, Linksys, MediaTone, MeetingPlace, MGX, Networkers, Networking Academy, Network Registrar, PCNow, PIX, PowerPanels, ProConnect, ScriptShare, SenderBase, SMARTnet, Spectrum Expert, StackWise, The Fastest Way to Increase Your Internet Quotient, TransPath, WebEx, and the WebEx logo are registered trademarks of Cisco Systems, Inc. and/or its affiliates in the United States and certain other countries.

All other trademarks mentioned in this document or Website 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. (0805R)

Introduction

Venture into any bookstore today and you can find numerous books on internetworking covering a wide range of topics from protocols to network design techniques. There's no question that internetworking has become a popular field with the enormous growth of the Internet and the increasing convergence of voice, video, and data. Cisco has built a very successful business selling the equipment that forms the network infrastructure—by some accounts, Cisco has more than 85 percent of the market—and at the same time has seen its Cisco IOS Software become a *de facto* industry standard. Yet, although plenty of material is written about network design and the protocols IOS supports, very little information is available from sources other than Cisco.

This lack of information is understandable—IOS is proprietary, after all—but it nevertheless leaves network implementers at a disadvantage. During our experience helping design and troubleshoot IOS-based networks, we've seen many cases where limited IOS architectural knowledge either contributed to a problem or made it more difficult to solve. In addition, we collectively have answered countless numbers of questions (and dispelled some myths) from bewildered Cisco customers about the workings of various IOS features.

This book is an attempt to bring together, in one place, the wealth of information about the architecture and the operation of IOS. Some of this information has been made public previously through forums, Cisco presentations, and the Cisco Technical Assistance Center. Most of the information you cannot find in the Cisco IOS documentation.

Objectives

Inside Cisco IOS Software Architecture is intended to be an IOS "shop manual" for network designers, implementers, and administrators. The objective of this book is to describe key parts of the architecture and the operation of the IOS software. This book also covers the architecture of some of Cisco's hardware platforms. Because IOS is a specialized embedded operating system tightly coupled to the underlying hardware, it's difficult to describe the software without also considering the hardware architecture. Note, however, that this book is not meant to be an exhaustive manual for Cisco hardware. The hardware descriptions are provided only to help illustrate unique features in the IOS software. You might notice that this book does not cover many of the Cisco platforms; in particular, this book does not cover any of the Catalyst switch products, and it omits many of the access routers. In most cases, the missing platforms either are similar to ones that are covered or, in the case of the Catalyst switches, would be best treated in a separate text of their own.

Organization

The book is divided into three general sections. The first covers the general architecture of IOS, including its software infrastructure and packet switching architecture. The second section, beginning with Chapter 3, examines the IOS implementations on a few selected Cisco hardware platforms, covering the design and operation of the platform-specific features. Finally, Chapter 8 describes how IOS implements select Quality of Service (QoS) mechanisms. The book is organized into the following chapters:

- **Chapter 1, "Fundamental IOS Software Architecture"**—Provides an introduction to operating system concepts and then covers the IOS software infrastructure, including processes, memory management, CPU scheduling, packet buffers, and device drivers.
- **Chapter 2**, **"Packet Switching Architecture"**—Gives an overview of the switching architecture and describes the theory of operation of several platform-independent switching methods, including process switching, fast switching, optimum switching, and Cisco Express Forwarding.
- **Chapter 3**, **"Shared Memory Routers"**—Shows how the features discussed in Chapter 1 and Chapter 2 actually are implemented on a platform using the relatively simple shared memory routers as an example.
- **Chapter 4, "Early Cbus Routers"**—Covers platform-specific switching features in the IOS implementation for the early Cbus routers. Describes the architecture of the AGS+ and Cisco 7000 routers.
- **Chapter 5, "Particle-Based Systems"**—Describes the particle-based packet buffering scheme using the Cisco 7200 router IOS implementation example. Covers the packet switching implementation on the Cisco 7200.
- **Chapter 6, "Cisco 7500 Routers"**—Continues the description of the Cbus architecture focusing on the Cisco 7500 router IOS implementation. Also examines the IOS distributed switching implementation on the Versatile Interface Processor (VIP).
- **Chapter 7, "The Cisco Gigabit Switch Router: 12000"**—Covers the IOS implementation on the Cisco 12000 series routers.
- **Chapter 8, "Quality of Service"**—Gives an overview of IOS Quality of Service (QoS) and describes several QoS methods, including priority queuing, custom queuing, weighted fair queuing, and modified deficit round robin.
- **Appendix A, "NetFlow Switching"**—Covers the NetFlow feature in IOS used for traffic monitoring and billing.

This chapter covers the following key topics:

- Hardware Architecture for Shared Memory Routers
- Packet Buffers for Shared Memory Routers
- Packet Switching on a Shared Memory Router

CHAPTER 3

Shared Memory Routers

In the previous two chapters, we covered IOS' basic architecture and switching methods in the abstract. To understand how IOS actually works, though, it's helpful to see how it operates on a real router. Because IOS is so closely coupled to the hardware on which it runs, each implementation has its own platform-specific features.

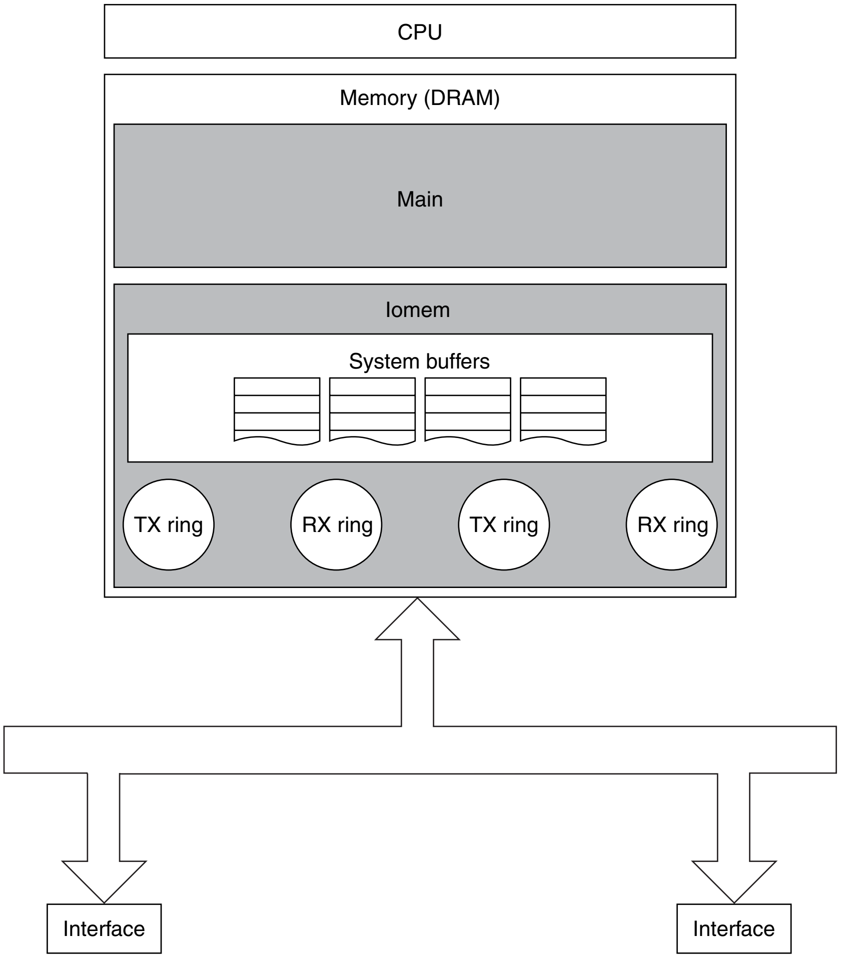
This chapter examines a very basic IOS implementation—the one for a group of routers known collectively as *shared memory* routers. Shared memory routers consist of several products, including the Cisco 1600, 2500, 4000, 4500, and 4700 series of routers. Although individually they all have their own unique features, together they have two major things in common: they all have a bare-bones architecture (just a CPU, main memory, and interfaces) and they all use the system buffers for packet buffering.

Hardware Architecture for Shared Memory Routers

Let's examine the hardware architecture on these shared memory platforms in some detail. We start with an overview of the generic shared memory architecture in Figure 3-1, and then mention some of the unique hardware features of platforms within this group.

You can see the overall architecture is fairly basic, consisting of just three major components: the processor (CPU), memory (DRAM), and interface controllers, with the processor and the interface controllers connected to the memory via data busses. Figure 3-1 also shows memory is divided into logical regions, which we'll examine later.

Figure 3-1 *Shared Memory Router Architecture*



CPU

The processor in a shared memory router is truly the brains behind the entire show; it runs IOS and switches packets. The processor is responsible for executing all the switching methods supported on these platforms; there are no offload processors involved.

The type of processor used depends on the platform. For example, the Cisco 1600 and 2500 series use a Motorola 68000 series CPU while the 4500 and 4700 series use a MIPS RISC CPU. You can determine the specific type of processor by looking at the output of the **show**

version command. Example 3-1 shows the **show version** output from a Cisco 1600 router. Example 3-2 shows the **show version** output from a Cisco 4500 router.

Example 3-1 **show version** *Output from a Cisco 1600 Router*

```
router-1600>show version
Cisco Internetwork Operating System Software
....
cisco 1604 (68360) processor (revision C) with 17920K/512K bytes of memory.
Processor board ID 05385389, with hardware revision 00972006
....
```

Example 3-2 **show version** *Output from a Cisco 4500 Router*

```
router-4500#show version
Cisco Internetwork Operating System Software
....
cisco 4500 (R4K) processor (revision B) with 16384K/16384K bytes of memory.
Processor board ID 01657190
R4600 processor, Implementation 32, Revision 2.0
....
```

Memory

Shared memory platforms use dynamic random access memory (DRAM) to hold most of the data in the router. DRAM contains all the routing tables, the caches, the IOS data, the packet buffers, and, on many systems, the IOS code itself.

IOS divides the available DRAM into two logical memory classes: *Local* and *Iomem* (I/O memory). In most cases, Local memory is placed into the **main** region and I/O memory is placed into the **iomem** region, as illustrated by the **show region** command output in Example 3-3.

Example 3-3 *DRAM Memory Regions Revealed in* **show region** *Command Output*

```
Router-4500#show region
Region Manager:

      Start         End     Size(b)  Class  Media  Name
 0x30000000  0x30FFFFFF    16777216  Flash  R/O    flash
 0x38000000  0x383FFFFF     4194304  Flash  R/O    bootflash
 0x40000000  0x40FFFFFF    16777216  Iomem  R/W    iomem
 0x60000000  0x61FFFFFF    33554432  Local  R/W    main
 0x600088A0  0x607229BF     7446816  IText  R/O    main:text
 0x60726000  0x6096506F     2355312  IData  R/W    main:data
 0x60965070  0x609DB1CF      483680  IBss   R/W    main:bss
 0x609DB1D0  0x61FFFFFF    23219760  Local  R/W    main:mainheap
 0x80000000  0x81FFFFFF    33554432  Local  R/W    main:(main_k0)
 0x88000000  0x88FFFFFF    16777216  Iomem  R/W    iomem:(iomem_k0)
 0xA0000000  0xA1FFFFFF    33554432  Local  R/W    main:(main_k1)
 0xA8000000  0xA8FFFFFF    16777216  Iomem  R/W    iomem:(iomem_k1)
```

The sizes of these two regions also are reflected in the output of **show version,** as illustrated in Example 3-4.

Example 3-4 **show version** *Command Output Also Reveals Memory Region Sizes*

```
Cisco Internetwork Operating System Software
IOS (tm) 4500 Software (C4500-J-M), Version 11.1(24), RELEASE SOFTWARE (fc1)
....
cisco 4500 (R4K) processor (revision E) with 32768K/16384K bytes of memory.
Processor board ID 09337282
....
```

The number before the slash is the amount of Local memory present (32,768 Kb), and the number after the slash is the amount of I/O memory present (16,384 Kb). The total DRAM memory present is the sum of these two numbers (49,152 Kb). For the system in Example 3-3 and Example 3-4, the 32,768 Kb of Local memory is assigned to the region named **main** and the 16,384 Kb of I/O memory is assigned to the region named **iomem**.

On shared memory platforms, the **main** region is used for storing routing tables, caches, general IOS data structures, and often the IOS runtime code—but not for storing packet buffers. Packet buffers are stored in the **iomem** region, which often is called *shared memory* because it's shared between the processor and the media interfaces.

On many systems, the DRAM used for Local memory is physically separate from the DRAM used for I/O memory—usually two different banks. On those systems, IOS simply assigns all the available memory in a bank to the appropriate memory regions, one bank for **iomem** and one bank for **main**. For the Cisco 1600 and 2500, however, I/O memory and Local memory are both allocated from the same DRAM bank. The amount of I/O memory carved from the installed DRAM depends on the total memory installed:

- **1MB**—512 Kb of memory is used for I/O
- **2MB**—1 MB of memory is used for I/O
- **4MB and above**—2 MB of memory is used for I/O

Location of IOS Runtime Code on Shared Memory Systems

Some IOS shared memory platforms—in particular, the 1600 series and the 2500 series routers—can run IOS directly from Flash memory. If a system is running IOS from Flash, it doesn't use the **main** memory region for the IOS runtime code. Instead, it stores the runtime code in a region named **flash**, as demonstrated in Example 3-5.

Example 3-5 **show region** *Output for a Run-from-Flash System*

```
Router-2500#show region
Region Manager:

      Start         End     Size(b)  Class  Media  Name
 0x00000000  0x007FFFFF     8388608  Local  R/W    main
 0x00001000  0x0001922F       98864  IData  R/W    main:data
 0x00019230  0x000666B3      316548  IBss   R/W    main:bss
 0x000666B4  0x007FEFFF     7965004  Local  R/W    main:heap
 0x007FF000  0x007FFFFF        4096  Local  R/W    main:flhlog
 0x00800000  0x009FFFFF     2097152  Iomem  R/W    iomem
 0x03000000  0x03FFFFFF    16777216  Flash  R/O    flash
 0x0304033C  0x037A7D37     7764476  IText  R/O    flash:text
```

In Chapter 1, "Fundamental IOS Software Architecture," you saw that the *IText* class is where the IOS runtime code is stored. In Example 3-5, the IText memory is assigned to a subregion of **flash** called **flash:text**, indicating the system is running IOS from Flash.

You also can determine whether a router is running IOS from Flash or Local memory by checking the image's filename. The last one or two letters of the filename indicate whether the image is *relocatable* or *run-from-DRAM*. The run-from-DRAM images run in Local memory, while the relocatable images run from Flash memory.

The **show version** output in Example 3-6 is taken from a Cisco 2500, which runs IOS from Flash.

Example 3-6 *Flash-based IOS Image*

```
router-2500>show version
Cisco Internetwork Operating System Software
IOS (tm) 2500 Software (C2500-JS-L), Version 12.0(7.3)T,  MAINTENANCE INTERIM SE
Copyright (c) 1986-1999 by cisco Systems, Inc.
....
```

The last letter in the image name (**L**) means this image is relocatable, and therefore is running from Flash. On the other hand, if the image runs from Local memory (DRAM), the image name ends in either **M** or **MZ**, as you can see in the output of **show version** in Example 3-7.

Example 3-7 *DRAM-based IOS Image*

```
Router-4500>show version
Cisco Internetwork Operating System Software
IOS (tm) 4500 Software (C4500-P-M), Version 11.2(18)P,  RELEASE SOFTWARE (fc1)
Copyright (c) 1986-1999 by cisco Systems, Inc.
....
```

Memory Pools

IOS creates two memory pools to manage allocation and de-allocation of memory within the DRAM regions. These two pools are named *Processor* and *I/O*, as demonstrated in the **show memory** output in Example 3-8.

Example 3-8 **show memory** *Output Reveals Memory Pools*

```
router-2500#show memory
                Head    Total(b)     Used(b)     Free(b)   Lowest(b) Largest(b)
Processor      3BB08    16528632      878596    15650036    15564280   15630436
      I/O    4000000     2097152      473468     1623684     1603060    1623232
```

The **Processor** memory pool is created from memory in the **main:heap** subregion of Local memory and the **I/O** pool is created from memory in the **iomem** region of I/O memory. The size of the **I/O** pool (the number in the **Total** column in Example 3-8) correlates closely to the size of **iomem**. However, the size of the **Processor** pool is always less than the size of the **main** memory region. The **Processor** pool gets only a subset of the memory in the **main** region because the remainder must be reserved for the IOS data, the BSS segments, and, on many platforms, the IOS runtime image itself.

NOTE Have you ever wondered why this collection of routers is called shared memory routers? All routers discussed in this book have shared memory, so why are these particular routers singled out? What's unique about these systems is not that they have shared memory, but that they have only one region of shared memory (I/O memory) and it's shared between a single main processor and all the interface controllers. In addition, that same memory is used for all packet switching—there's no data copied between regions to pass a packet from fast to process switching like there is on other platforms.

Interface Controllers

Interface controllers are responsible for transferring packets to and from the physical media. They have their own processors, called media controllers, but do not perform any switching or IOS processing operations. Interface controllers perform media control operations and move packets between I/O memory and the network media. Depending on the platform, interface controllers can be implemented as removable port modules or as components on the main system board.

Packet Buffers for Shared Memory Routers

You might recall from Chapter 1 that IOS maintains a set of packet buffers called *system buffers* that are used primarily for process switching packets. IOS on shared memory routers also uses system buffers, but in a distinct way—it uses the system buffers for *all* packet switching, not just process switching.

In addition to the standard public buffer pools, IOS on shared memory platforms also creates some private system buffer pools and special buffer structures for the interface controllers called *rings*.

Private Buffer Pools

Private buffer pools are used for packet switching just like the public pools, but are intended to prevent interface buffer starvation. All interfaces must compete for buffers from the public pools, increasing the probability for buffer contention (which degrades performance) and raising the possibility that a single interface can starve out others needing buffers from a single pool. With the addition of private pools, each interface is given a number of buffers dedicated for its use, so contention is minimized. Although most interfaces receive their own private pool of buffers, some lower speed interfaces, such as asynchronous interfaces, do not.

Unlike the dynamic public pools, the private pools are static and are allocated with a fixed number of buffers at IOS initialization. New buffers cannot be created on demand for these pools. If a buffer is needed and one is not available in the private pool, IOS *falls back* to the public buffer pool for the size that matches the interface's MTU.

The output of the **show buffers** command in Example 3-9 shows both the public and the private pools.

Example 3-9 **show buffers** *Command Output*

```
Router#show buffers
Buffer elements:
     500 in free list (500 max allowed)
     57288024 hits, 0 misses, 0 created

Public buffer pools:
Small buffers, 104 bytes (total 50, permanent 50):
     50 in free list (20 min, 150 max allowed)
     11256002 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
Middle buffers, 600 bytes (total 25, permanent 25):
     24 in free list (10 min, 150 max allowed)
     3660412 hits, 12 misses, 36 trims, 36 created
     0 failures (0 no memory)
```

continues

Example 3-9 **show buffers** *Command Output (Continued)*

```
Big buffers, 1524 bytes (total 50, permanent 50):
     50 in free list (5 min, 150 max allowed)
     585512 hits, 14 misses, 12 trims, 12 created
     2 failures (0 no memory)
VeryBig buffers, 4520 bytes (total 10, permanent 10):
     10 in free list (0 min, 100 max allowed)
     0 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
Large buffers, 5024 bytes (total 0, permanent 0):
     0 in free list (0 min, 10 max allowed)
     0 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
Huge buffers, 18024 bytes (total 0, permanent 0):
     0 in free list (0 min, 4 max allowed)
     0 hits, 0 misses, 0 trims, 0 created
     0 failures (0 no memory)
Interface buffer pools:
Ethernet0 buffers, 1524 bytes (total 32, permanent 32):
     8 in free list (0 min, 32 max allowed)
     3398 hits, 3164 fallbacks
     8 max cache size, 7 in cache
Serial0 buffers, 1524 bytes (total 32, permanent 32):
     7 in free list (0 min, 32 max allowed)
     25 hits, 0 fallbacks
     8 max cache size, 8 in cache
Serial1 buffers, 1524 bytes (total 32, permanent 32):
     7 in free list (0 min, 32 max allowed)
     25 hits, 0 fallbacks
     8 max cache size, 8 in cache
```

Although most of these fields are explained in Chapter 1, some are unique to the interface buffer pools:

- **fallbacks**—The number of times the interface processor had to fall back to the public buffer pools to find a buffer in which to store a packet.
- **max cache size**—Some number of buffers in each private buffer pool are cached for faster access; this is the maximum number of buffers that can be cached.
- **in cache**—The number of cached private buffers.

Notice the private pools do not have a **creates** or a **trims** field; this is because these pools are static pools.

Receive Rings and Transmit Rings

In addition to public and private buffer pools, IOS also creates special buffer control structures, called rings, in I/O memory. IOS and interface controllers use these rings to control which buffers are used to receive and transmit packets to the media. Rings are actually a common control structure used by many types of media controllers to manage the memory for packets being received or waiting to be transmitted. The rings themselves consist of media controller–specific elements that point to individual packet buffers elsewhere in I/O memory. IOS creates these rings on behalf of the media controllers and then manages them jointly with the controllers.

Each interface has a pair of rings: a receive ring for receiving packets and a transmit ring for transmitting packets. These rings have fixed sizes determined by several factors. The size of the receive ring depends on the specific interface and is dictated by the media controller specification. The transmit ring size, however, is dependent on the media controller specification and the type of queuing configured on the interface.

Receive rings have a constant number of packet buffers allocated to them that equals the size of the ring. The receive ring's packet buffers initially are allocated from the interface's private buffer pool. During operation, they might be replaced with buffers from either the private pool or a public pool.

The number of buffers allocated to a transmit ring can vary from zero up to the maximum size of the transmit ring. Transmit ring packet buffers come from the receive ring of the originating interface for a switched packet or from a public pool if the packet was originated by IOS. They're de-allocated from the transmit ring and returned to their original pool after the payload data is transmitted.

The **show controller** command in Example 3-10 displays the sizes and the locations of the receive and the transmit rings. Although most interface types produce output similar to what's shown, the exact content of the output varies.

Example 3-10 **show controller** *Command Output Displays Receive and Transmit Ring Sizes and Locations*

```
isp-4700b#show controller ethernet
AM79970 unit 0 NIM slot 1, NIM type code 14, NIM version 1
Media Type is 10BaseT, Half Duplex, Link State is Down, Squelch is Normal
idb 0x60AE8324, ds 0x60AE9D10, eim_regs = 0x3C110000
IB at 0x40006E64: mode=0x0010, mcfilter 0000/0000/0100/0000
station address 0060.837c.7089  default station address 0060.837c.7089
buffer size 1524
RX ring with 32 entries at 0x400303D0
Rxhead = 0x400303D0 (0), Rxp = 0x60AE9D28 (0)
00 pak=0x60AF2468 ds=0xA80C745E status=0x80 max_size=1524 pak_size=0
01 pak=0x60AF2254 ds=0xA80C6DA2 status=0x80 max_size=1524 pak_size=0
02 pak=0x60AF2040 ds=0xA80C66E6 status=0x80 max_size=1524 pak_size=0
....
```

continues

Example 3-10 **show controller** *Command Output Displays Receive and Transmit Ring Sizes and Locations (Continued)*

```
TX ring with 32 entries at 0x40059BD0, tx_count = 0
tx_head = 0x40059BD0 (0), head_txp = 0x60AE9E3C (0)
tx_tail = 0x40059BD0 (0), tail_txp = 0x60AE9E3C (0)
00 pak=0x000000 ds=0xA8000000 status=0x03 status2=0x0000 pak_size=0
01 pak=0x000000 ds=0xA8000000 status=0x03 status2=0x0000 pak_size=0
02 pak=0x000000 ds=0xA8000000 status=0x03 status2=0x0000 pak_size=0
```

The following list describes some of the more interesting fields in the **show controller** output from Example 3-10:

- **RX ring with 32 entries at 0x400303D0**—The size of the receive ring is 32 and it begins at memory location 0x400303D0 in I/O memory.
- **00 pak=0x60AF2468 ds=0xA80C745E status=0x80 max_size=1524 pak_size=0**—This line is repeated for each ring entry in the receive ring. Each ring entry, called a *descriptor*, contains information about a corresponding packet buffer allocated to the receive ring. All the fields in the descriptor except for **pak** are device specific and vary from media controller to media controller. The **pak** field points to the memory address of the *header* for the packet linked to this descriptor. All IOS packet buffers have a corresponding header containing information about the contents of the buffer and a pointer to the actual location of the buffer itself. Although packet buffers on shared memory systems are located in I/O memory, their headers might reside in Local memory, as is the case here.
- **TX ring with 32 entries at 0x40059BD0, tx_count = 0**—Shows the transmit ring size and the number of packets waiting to be transmitted. In this case, the transmit ring size is 32 and there are no packets awaiting transmission on this interface.
- **00 pak=0x000000 ds=0xA8000000 status=0x03 status2=0x0000 pak_size=0**—This line is repeated for each entry in the transmit ring. Like the receive ring, each entry in the transmit ring is a descriptor containing information about a corresponding packet buffer on the ring. The **pak** field points to the header of a corresponding packet buffer waiting to be transmitted. The other fields are media controller specific. Unlike receive ring descriptors, transmit ring descriptors are linked only to a packet buffer when there is a packet waiting to transmit. After a packet buffer is transmitted, the buffer is unlinked from the transmit descriptor, returned to its original free pool, and its descriptor **pak** pointer reset to 0x000000.

Packet Switching on a Shared Memory Router

Now that we've looked at the general hardware architecture of the shared memory routers and how IOS divides their memory, let's look at how IOS actually switches packets. IOS on shared memory routers supports the following:

- Process switching
- CEF switching
- Fast switching

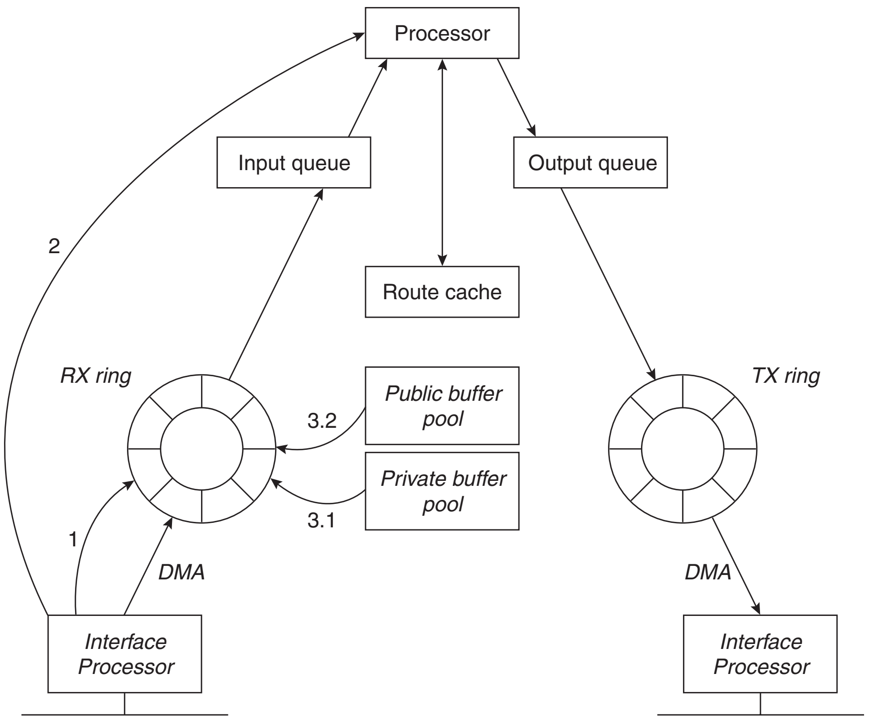
There are three stages in IOS packet switching:

1. Receiving the packet
2. Switching the packet
3. Transmitting the packet

Receiving the Packet

Figure 3-2 illustrates the steps of the packet receive stage.

Figure 3-2 *Receiving the Packet*



Step 1 The interface media controller detects a packet on the network media and copies it into a buffer pointed to by the first free element in the receive ring (that is, the next buffer the media controller owns). Media controllers use the DMA (direct memory access) method to copy packet data into memory.

Step 2 The media controller changes ownership of the packet buffer back to the processor and issues a receive interrupt to the processor. The media controller does not have to wait for a response from the CPU and continues to receive incoming packets into other buffers linked to the receive ring.

NOTE Notice in Step 2 the media controller can continue to receive incoming packets into the receive ring. Therefore, it's possible for the media controller to fill the receive ring before the processor processes all the new buffers in the ring. This condition is called an *overrun*; when it occurs, all incoming packets are dropped until the processor catches up.

Step 3 The CPU responds to the receive interrupt, and then attempts to remove the newly-filled buffer from the receive ring and replenish the ring from the interface's private pool. Three outcomes are possible:

3.1 A free buffer is available in the interface's private pool to replenish the receive ring: The free buffer is linked to the receive ring and packet switching continues with Step 4.

3.2 A free buffer is not available in the interface's private pool, so the receive ring is replenished by *falling back* to the global pool that matches the interface's MTU. The *fallback* counter is incremented for the private pool.

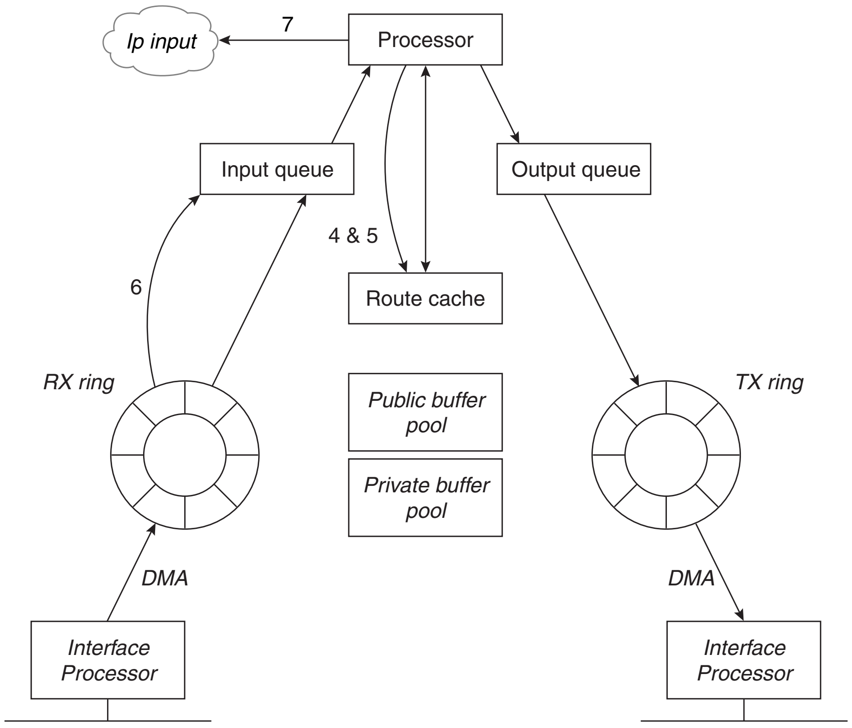
3.3 If a free buffer is not available in the public pool as well, the incoming packet is dropped and the *ignore* counter is incremented. Further, the interface is *throttled*, and all incoming traffic is ignored on this interface for some short period of time.

Switching the Packet

Step 4 After the receive ring is replenished, the CPU begins actually switching the packet. This operation consists of locating information about where to send the packet (next hop) and the MAC header to rewrite onto the packet.

IOS attempts to switch the packet using the fastest method configured on the interface. On shared memory systems, it first tries CEF switching (if configured), then fast switching, and finally falls back to process switching if none of the others work. Figure 3-3 illustrates the steps of the packet switching stage and the list that follows describes the steps illustrated in the figure.

Figure 3-3 *Switching the Packet*



Step 5 While still in the receive interrupt context, IOS attempts to use the CEF table (first) or fast switching cache (second) to make a switching decision.

- **5.1** **CEF switching**—If CEF switching is enabled on the interface, then CEF switching is attempted. Four different outcomes are possible:
  - ***5.1.1*** *If there are valid CEF and adjacency table entries, IOS rewrites the MAC header on the packet and begins transmitting it in Step 8, the packet transmit stage.*
  - ***5.1.2*** *If the CEF adjacency entry for this destination points to a punt adjacency, IOS proceeds to try fast switching (see Step 5.2).*
  - ***5.1.3*** *If the CEF table entry for this packet points to a receive adjacency, the packet is queued for process switching.*
  - ***5.1.4*** *If there is no CEF entry for the destination, the packet is dropped.*
- **5.2** **Fast switching**—If CEF is not enabled or the packet cannot be CEF switched, IOS attempts to fast switch the packet.
  - ***5.2.1*** *If there is a valid fast cache entry for this destination, IOS rewrites the MAC header information and begins transmitting the packet (Step 8, packet transmit stage).*
  - ***5.2.2*** *If there is no valid fast cache entry, the packet is queued for process switching (Step 6).*

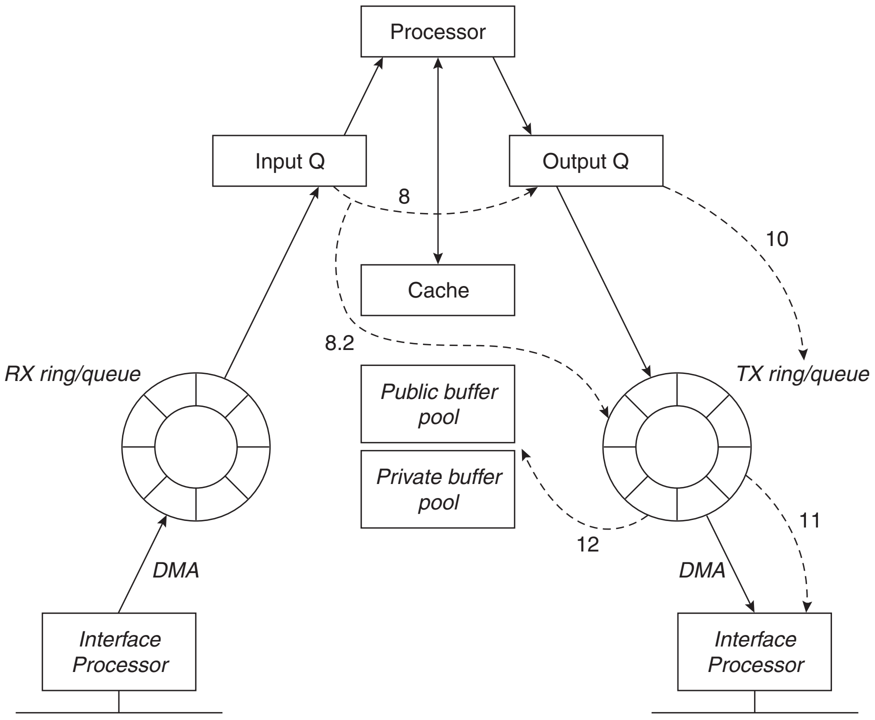
Step 6 **Process switching**—If both CEF switching and fast switching fail, IOS falls back to process switching. The packet is placed in the input queue of the appropriate process (an IP packet is placed in the queue for the IP Input process, for instance), and the receive interrupt is dismissed.

Step 7 Eventually, the packet switching process runs, switching the packet and rewriting the MAC header as needed. Note the packet still has not moved from the buffer it was originally copied into. After the packet is switched, IOS continues to the packet transmit stage, for process switching (Step 9).

Transmitting the Packet

Figure 3-4 illustrates the steps of packet transmit stage.

Figure 3-4 *Transmitting the Packet*



Step 8 If the packet was CEF switched or fast switched, then while still in receive interrupt context IOS checks to see if there are packets on the output queue of the outbound interface.

8.1 If there are packets already on the output hold queue for the interface, IOS places the packet on the output hold queue instead of directly into the transmit ring to reduce the possibility of out-of-order packets, and then proceeds to Step 8.3.

8.2 If the output hold queue is empty, IOS places the packet on the transmit ring of the output interface by linking the packet buffer to a transmit ring descriptor. The receive interrupt is dismissed and processing continues with Step 11. If there is no room on the transmit ring, the packet is placed on the output hold queue instead and the receive interrupt is dismissed.

8.3 If the output hold queue is full, the packet is dropped, the output **drop** counter is incremented, and the receive interrupt is dismissed.

NOTE In Step 8 of the transmit stage, notice that IOS checks the output hold queue first before placing any CEF or fast switched packets on the transmit ring. This reduces the possibility of transmitting packets out of order. How can packets get out of order when they're being switched as they're received? Consider the following example.

Let's assume the first packet in a given conversation (flow) between two hosts arrives at the router. IOS attempts to fast switch the packet and finds there is no fast cache entry. So, the packet is handed off to the IP Input process for process switching.

So far, everything is fine. In the process of switching the packet, IP Input builds a cache entry for this flow of packets and places the first packet on the output queue of the outbound interface.

Now, let's assume that just at this moment a second packet arrives for the same flow. IOS fast switches the second packet (because the cache entry has now been built) and gets ready to transmit it. If IOS puts this second packet directly on the transmit ring, it is transmitted before the first packet, which is still waiting in the output queue. To prevent this situation from occurring, IOS always makes certain the output queue of the outbound interface is empty before placing any CEF-switched or fast-switched packets directly on the transmit ring.

Step 9 If the packet was process switched, the packet is placed on the output queue for the output interface. If the output queue is full, the packet is dropped and the *output drop* counter is incremented.

Step 10 IOS attempts to find a free descriptor in the output interface transmit ring. If a free descriptor exists, IOS removes the packet from the output hold queue and links the buffer to the transmit ring. If no free descriptor exists (that is, the ring is full), IOS leaves the packet in the output hold queue until the media controller transmits a packet from the ring and frees a descriptor.

Step 11 The outbound interface media controller polls its transmit ring periodically for packets that need to be transmitted. As soon as the media controller detects a packet, it copies the packet onto the network media and raises a transmit interrupt to the processor.

Step 12 IOS acknowledges the transmit interrupt, unlinks the packet buffer from the transmit ring, and returns the buffer to the pool of buffers from which it originally came. IOS then checks the output hold queue for the interface; if there are any packets waiting in the output hold queue, IOS removes the next one from the queue and links it to the transmit ring. Finally, the transmit interrupt is dismissed.

Summary

This chapter covered the IOS implementation on Cisco's shared memory architecture routers, including the 2500 series, of which there are more than 1 million in use today. You saw how IOS divides memory into regions and how it actually switches packets on these platforms.

INDEX