

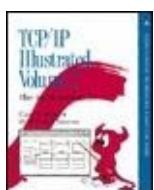
## **TCP/IP Illustrated, Volume 2: The Implementation**

By [Gary R. Wright, W. Richard Stevens](#)

[Start Reading ▶](#)

Publisher : Addison Wesley  
Pub Date : January 12, 1995  
ISBN : 0-201-63354-X  
Pages : 1200

*TCP/IP Illustrated, Volume 2* contains a thorough explanation of how TCP/IP protocols are implemented. There isn't a more practical or up-to-date book this volume is the only one to cover the de facto standard implementation from the 4.4BSD-Lite release, the foundation for TCP/IP implementations run daily on hundreds of thousands of systems worldwide.

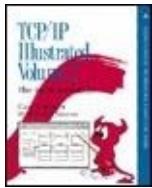


- [Table of Contents](#)

Combining 500 illustrations with 15,000 lines of real, working code, *TCP/IP Illustrated, Volume 2* uses a teach-by-example approach to help you master TCP/IP implementation. You will learn about such topics as the

relationship between the sockets API and the protocol suite, and the differences between a host implementation and a router. In addition, the book covers the newest features of the 4.4BSD-Lite release, including multicasting, long fat pipe support, window scale, timestamp options, and protection against wrapped sequence numbers, and many other topics.

Comprehensive in scope, based on a working standard, and thoroughly illustrated, this book is an indispensable resource for anyone working with TCP/IP.



## TCP/IP Illustrated, Volume 2: The Implementation

By [Gary R. Wright, W. Richard Stevens](#)

[Start Reading ▶](#)

Publisher : Addison Wesley  
Pub Date : January 12, 1995  
ISBN : 0-201-63354-X  
Pages : 1200

- [Table of Contents](#)

[Copyright](#)

[Preface](#)

[Introduction](#)

[Organization of the Book](#)

[Intended Audience](#)

[Source Code Copyright](#)

[Acknowledgments](#)

[Chapter 1. Introduction](#)

[Section 1.1. Introduction](#)

[Section 1.2. Source Code Presentation](#)

[Section 1.3. History](#)

[Section 1.4. Application Programming Interfaces](#)

[Section 1.5. Example Program](#)

[Section 1.6. System Calls and Library Functions](#)

[Section 1.7. Network Implementation Overview](#)

[Section 1.8. Descriptors](#)

[Section 1.9. Mbufs \(Memory Buffers\) and Output Processing](#)

[Section 1.10. Input Processing](#)

[Section 1.11. Network Implementation Overview Revisited](#)

[Section 1.12. Interrupt Levels and Concurrency](#)

[Section 1.13. Source Code Organization](#)

[Section 1.14. Test Network](#)

[Section 1.15. Summary](#)

[Chapter 2. Mbufs: Memory Buffers](#)

[Section 2.1. Introduction](#)

[Section 2.2. Code Introduction](#)

[Section 2.3. Mbuf Definitions](#)

[Section 2.4. mbuf Structure](#)

[Section 2.5. Simple Mbuf Macros and Functions](#)

[Section 2.6. m\\_devget and m\\_pullup Functions](#)

[Section 2.7. Summary of Mbuf Macros and Functions](#)

Section 2.8. Summary of Net/3 Networking Data Structures

Section 2.9. m\_copy and Cluster Reference Counts

Section 2.10. Alternatives

Section 2.11. Summary

Chapter 3. Interface Layer

Section 3.1. Introduction

Section 3.2. Code Introduction

Section 3.3. ifnet Structure

Section 3.4. ifaddr Structure

Section 3.5. sockaddr Structure

Section 3.6. ifnet and ifaddr Specialization

Section 3.7. Network Initialization Overview

Section 3.8. Ethernet Initialization

Section 3.9. SLIP Initialization

Section 3.10. Loopback Initialization

Section 3.11. if\_attach Function

Section 3.12. ifinit Function

3.13 Summary

Chapter 4. Interfaces: Ethernet

Section 4.1. Introduction

Section 4.2. Code Introduction

Section 4.3. Ethernet Interface

Section 4.4. ioctl System Call

Section 4.5. Summary

Chapter 5. Interfaces: SLIP and Loopback

Section 5.1. Introduction

Section 5.2. Code Introduction

Section 5.3. SLIP Interface

Section 5.4. Loopback Interface

Section 5.5. Summary

Chapter 6. IP Addressing

Section 6.1. Introduction

Section 6.2. Code Introduction

Section 6.3. Interface and Address Summary

Section 6.4. sockaddr\_in Structure

Section 6.5. in\_ifaddr Structure

Section 6.6. Address Assignment

Section 6.7. Interface ioctl Processing

Section 6.8. Internet Utility Functions

Section 6.9. ifnet Utility Functions

Section 6.10. Summary

Chapter 7. Domains and Protocols

Section 7.1. Introduction

- Section 7.2. Code Introduction
- Section 7.3. domain Structure
- Section 7.4. protosw Structure
- Section 7.5. IP domain and protosw Structures
- Section 7.6. pffindproto and pffindtype Functions
- Section 7.7. pfctlinput Function
- Section 7.8. IP Initialization
- Section 7.9. sysctl System Call
- Section 7.10. Summary

## Chapter 8. IP: Internet Protocol

- Section 8.1. Introduction
- Section 8.2. Code Introduction
- Section 8.3. IP Packets
- Section 8.4. Input Processing: ipintr Function
- Section 8.5. Forwarding: ip\_forward Function
- Section 8.6. Output Processing: ip\_output Function
- Section 8.7. Internet Checksum: in\_cksum Function
- Section 8.8. setsockopt and getsockopt System Calls
- Section 8.9. ip\_sysctl Function
- Section 8.10. Summary

## Chapter 9. IP Option Processing

- Section 9.1. Introduction
- Section 9.2. Code Introduction
- Section 9.3. Option Format
- Section 9.4. ip\_dooptions Function
- Section 9.5. Record Route Option
- Section 9.6. Source and Record Route Options
- Section 9.7. Timestamp Option
- Section 9.8. ip\_insertoptions Function
- Section 9.9. ip\_pcbopts Function
- Section 9.10. Limitations
- Section 9.11. Summary

## Chapter 10. IP Fragmentation and Reassembly

- Section 10.1. Introduction
- Section 10.2. Code Introduction
- Section 10.3. Fragmentation
- Section 10.4. ip\_optcopy Function
- Section 10.5. Reassembly
- Section 10.6. ip\_reass Function
- Section 10.7. ip\_slowtimo Function
- Section 10.8. Summary

## Chapter 11. ICMP: Internet Control Message Protocol

- Section 11.1. Introduction
- Section 11.2. Code Introduction

- Section 11.3. icmp Structure
- Section 11.4. ICMP protosw Structure
- Section 11.5. Input Processing: icmp\_input Function
- Section 11.6. Error Processing
- Section 11.7. Request Processing
- Section 11.8. Redirect Processing
- Section 11.9. Reply Processing
- Section 11.10. Output Processing
- Section 11.11. icmp\_error Function
- Section 11.12. icmp\_reflect Function
- Section 11.13. icmp\_send Function
- Section 11.14. icmp\_sysctl Function
- Section 11.15. Summary

## Chapter 12. IP Multicasting

- Section 12.1. Introduction
- Section 12.2. Code Introduction
- Section 12.3. Ethernet Multicast Addresses
- Section 12.4. ether\_multi Structure
- Section 12.5. Ethernet Multicast Reception
- Section 12.6. in\_multi Structure
- Section 12.7. ip\_moptions Structure
- Section 12.8. Multicast Socket Options
- Section 12.9. Multicast TTL Values
- Section 12.10. ip\_setmoptions Function
- Section 12.11. Joining an IP Multicast Group
- Section 12.12. Leaving an IP Multicast Group
- Section 12.13. ip\_getmoptions Function
- Section 12.14. Multicast Input Processing: ipintr Function
- Section 12.15. Multicast Output Processing: ip\_output Function
- Section 12.16. Performance Considerations
- Section 12.17. Summary

## Chapter 13. IGMP: Internet Group Management Protocol

- Section 13.1. Introduction
- Section 13.2. Code Introduction
- Section 13.3. igmp Structure
- Section 13.4. IGMP protosw Structure
- Section 13.5. Joining a Group: igmp\_joingroup Function
- Section 13.6. igmp\_fasttimout Function
- Section 13.7. Input Processing: igmp\_input Function
- Section 13.8. Leaving a Group: igmp\_leavegroup Function
- Section 13.9. Summary

## Chapter 14. IP Multicast Routing

- Section 14.1. Introduction
- Section 14.2. Code Introduction

- Section 14.3. Multicast Output Processing Revisited
  - Section 14.4. mrouted Daemon
  - Section 14.5. Virtual Interfaces
  - Section 14.6. IGMP Revisited
  - Section 14.7. Multicast Routing
  - Section 14.8. Multicast Forwarding: ip\_mforward Function
  - Section 14.9. Cleanup: ip\_mrouter\_done Function
  - Section 14.10. Summary
- Chapter 15. Socket Layer
- Section 15.1. Introduction
  - Section 15.2. Code Introduction
  - Section 15.3. socket Structure
  - Section 15.4. System Calls
  - Section 15.5. Processes, Descriptors, and Sockets
  - Section 15.6. socket System Call
  - Section 15.7. getsock and sockargs Functions
  - Section 15.8. bind System Call
  - Section 15.9. listen System Call
  - Section 15.10. tsleep and wakeup Functions
  - Section 15.11. accept System Call
  - Section 15.12. sonewconn and soisconnected Functions
  - Section 15.13. connect System call
  - Section 15.14. shutdown System Call
  - Section 15.15. close System Call
  - Section 15.16. Summary
- Chapter 16. Socket I/O
- Section 16.1. Introduction
  - Section 16.2. Code Introduction
  - Section 16.3. Socket Buffers
  - Section 16.4. write, writev, sendto, and sendmsg System Calls
  - Section 16.5. sendmsg System Call
  - Section 16.6. sendit Function
  - Section 16.7. sosend Function
  - Section 16.8. read, readyv, recvfrom, and recvmsg System Calls
  - Section 16.9. recvmsg System Call
  - Section 16.10. recvv Function
  - Section 16.11. soreceive Function
  - Section 16.12. soreceive Code
  - Section 16.13. select System Call
  - Section 16.14. Summary
- Chapter 17. Socket Options
- Section 17.1. Introduction
  - Section 17.2. Code Introduction
  - Section 17.3. setsockopt System Call

- Section 17.4. `getsockopt` System Call
  - Section 17.5. `fcntl` and `ioctl` System Calls
  - Section 17.6. `getsockname` System Call
  - Section 17.7. `getpeername` System Call
  - Section 17.8. Summary
- Chapter 18. Radix Tree Routing Tables
- Section 18.1. Introduction
  - Section 18.2. Routing Table Structure
  - Section 18.3. Routing Sockets
  - Section 18.4. Code Introduction
  - Section 18.5. Radix Node Data Structures
  - Section 18.6. Routing Structures
  - Section 18.7. Initialization: `route_init` and `rtable_init` Functions
  - Section 18.8. Initialization: `rn_init` and `rn_inithead` Functions
  - Section 18.9. Duplicate Keys and Mask Lists
  - Section 18.10. `rn_match` Function
  - Section 18.11. `rn_search` Function
  - Section 18.12. Summary
- Chapter 19. Routing Requests and Routing Messages
- Section 19.1. Introduction
  - Section 19.2. `rtalloc` and `rtalloc1` Functions
  - Section 19.3. `RTFREE` Macro and `rtfree` Function
  - Section 19.4. `rtrequest` Function
  - Section 19.5. `rt_setgate` Function
  - Section 19.6. `rtinit` Function
  - Section 19.7. `rtredirect` Function
  - Section 19.8. Routing Message Structures
  - Section 19.9. `rt_missmsg` Function
  - Section 19.10. `rt_ifmsg` Function
  - Section 19.11. `rt_newaddrmsg` Function
  - Section 19.12. `rt_msg1` Function
  - Section 19.13. `rt_msg2` Function
  - Section 19.14. `sysctl_rtable` Function
  - Section 19.15. `sysctl_dumpentry` Function
  - Section 19.16. `sysctl_iflist` Function
  - Section 19.17. Summary
- Chapter 20. Routing Sockets
- Section 20.1. Introduction
  - Section 20.2. `routedomain` and `protosw` Structures
  - Section 20.3. Routing Control Blocks
  - Section 20.4. `raw_init` Function
  - Section 20.5. `route_output` Function
  - Section 20.6. `rt_xaddrs` Function
  - Section 20.7. `rt_setmetrics` Function

- Section 20.8. `raw_input` Function
  - Section 20.9. `route_usrreq` Function
  - Section 20.10. `raw_usrreq` Function
  - Section 20.11. `raw_attach`, `raw_detach`, and `raw_disconnect` Functions
  - Section 20.12. Summary
- Chapter 21. ARP: Address Resolution Protocol
- Section 21.1. Introduction
  - Section 21.2. ARP and the Routing Table
  - Section 21.3. Code Introduction
  - Section 21.4. ARP Structures
  - Section 21.5. `arpwhohas` Function
  - Section 21.6. `arprequest` Function
  - Section 21.7. `arpintr` Function
  - Section 21.8. `in_arpinput` Function
  - Section 21.9. ARP Timer Functions
  - Section 21.10. `arpresolve` Function
  - Section 21.11. `arplookup` Function
  - Section 21.12. Proxy ARP
  - Section 21.13. `arp_rtrequest` Function
  - Section 21.14. ARP and Multicasting
  - Section 21.15. Summary
- Chapter 22. Protocol Control Blocks
- Section 22.1. Introduction
  - Section 22.2. Code Introduction
  - Section 22.3. `inpcb` Structure
  - Section 22.4. `in_pcbaalloc` and `in_pcbadetach` Functions
  - Section 22.5. Binding, Connecting, and Demultiplexing
  - Section 22.6. `in_pcblklookup` Function
  - Section 22.7. `in_pcbbind` Function
  - Section 22.8. `in_pcbbconnect` Function
  - Section 22.9. `in_pcbadisconnect` Function
  - Section 22.10. `in_setsockaddr` and `in_setpeeraddr` Functions
  - Section 22.11. `in_pcbanotify`, `in_rtchange`, and `in_losing` Functions
  - Section 22.12. Implementation Refinements
  - Section 22.13. Summary
- Chapter 23. UDP: User Datagram Protocol
- Section 23.1. Introduction
  - Section 23.2. Code Introduction
  - Section 23.3. UDP protosw Structure
  - Section 23.4. UDP Header
  - Section 23.5. `udp_init` Function
  - Section 23.6. `udp_output` Function
  - Section 23.7. `udp_input` Function
  - Section 23.8. `udp_saveopt` Function

- Section 23.9. `udp_ctlinput` Function
  - Section 23.10. `udp_usrreq` Function
  - Section 23.11. `udp_sysctl` Function
  - Section 23.12. Implementation Refinements
  - Section 23.13. Summary
- Chapter 24. TCP: Transmission Control Protocol
- Section 24.1. Introduction
  - Section 24.2. Code Introduction
  - Section 24.3. TCP protosw Structure
  - Section 24.4. TCP Header
  - Section 24.5. TCP Control Block
  - Section 24.6. TCP State Transition Diagram
  - Section 24.7. TCP Sequence Numbers
  - Section 24.8. `tcp_init` Function
  - Section 24.9. Summary
- Chapter 25. TCP Timers
- Section 25.1. Introduction
  - Section 25.2. Code Introduction
  - Section 25.3. `tcp_canceltimers` Function
  - Section 25.4. `tcp_fasttimo` Function
  - Section 25.5. `tcp_slowtimo` Function
  - Section 25.6. `tcp_timers` Function
  - Section 25.7. Retransmission Timer Calculations
  - Section 25.8. `tcp_newtcpcb` Function
  - Section 25.9. `tcp_setpersist` Function
  - Section 25.10. `tcp_xmit_timer` Function
  - Section 25.11. Retransmission Timeout: `tcp_timers` Function
  - Section 25.12. An RTT Example
  - Section 25.13. Summary
- Chapter 26. TCP Output
- Section 26.1. Introduction
  - Section 26.2. `tcp_output` Overview
  - Section 26.3. Determine if a Segment Should be Sent
  - Section 26.4. TCP Options
  - Section 26.5. Window Scale Option
  - Section 26.6. Timestamp Option
  - Section 26.7. Send a Segment
  - Section 26.8. `tcp_template` Function
  - Section 26.9. `tcp_respond` Function
  - Section 26.10. Summary
- Chapter 27. TCP Functions
- Section 27.1. Introduction
  - Section 27.2. `tcp_drain` Function
  - Section 27.3. `tcp_drop` Function

- Section 27.4. `tcp_close` Function
- Section 27.5. `tcp_mss` Function
- Section 27.6. `tcp_ctlinput` Function
- Section 27.7. `tcp_notify` Function
- Section 27.8. `tcp_quench` Function
- Section 27.9. `TCP_REASS` Macro and `tcp_reass` Function
- Section 27.10. `tcp_trace` Function
- Section 27.11. Summary

## Chapter 28. TCP Input

- Section 28.1. Introduction
- Section 28.2. Preliminary Processing
- Section 28.3. `tcp_dooptions` Function
- Section 28.4. Header Prediction
- Section 28.5. TCP Input: Slow Path Processing
- Section 28.6. Initiation of Passive Open, Completion of Active Open
- Section 28.7. PAWS: Protection Against Wrapped Sequence Numbers
- Section 28.8. Trim Segment so Data is Within Window
- Section 28.9. Self-Connects and Simultaneous Opens
- Section 28.10. Record Timestamp
- Section 28.11. RST Processing
- Section 28.12. Summary

## Chapter 29. TCP Input (Continued)

- Section 29.1. Introduction
- Section 29.2. ACK Processing Overview
- Section 29.3. Completion of Passive Opens and Simultaneous Opens
- Section 29.4. Fast Retransmit and Fast Recovery Algorithms
- Section 29.5. ACK Processing
- Section 29.6. Update Window Information
- Section 29.7. Urgent Mode Processing
- Section 29.8. `tcp_pullofband` Function
- Section 29.9. Processing of Received Data
- Section 29.10. FIN Processing
- Section 29.11. Final Processing
- Section 29.12. Implementation Refinements
- Section 29.13. Header Compression
- Section 29.14. Summary

## Chapter 30. TCP User Requests

- Section 30.1. Introduction
- Section 30.2. `tcp_usrreq` Function
- Section 30.3. `tcp_attach` Function
- Section 30.4. `tcp_disconnect` Function
- Section 30.5. `tcp_usrclosed` Function
- Section 30.6. `tcp_ctloutput` Function
- Section 30.7. Summary

**Chapter 31. BPF: BSD Packet Filter**

- Section 31.1. Introduction
- Section 31.2. Code Introduction
- Section 31.3. bpf\_if Structure
- Section 31.4. bpf\_d Structure
- Section 31.5. BPF Input
- Section 31.6. BPF Output
- Section 31.7. Summary

**Chapter 32. Raw IP**

- Section 32.1. Introduction
- Section 32.2. Code Introduction
- Section 32.3. Raw IP protosw Structure
- Section 32.4. rip\_init Function
- Section 32.5. rip\_input Function
- Section 32.6. rip\_output Function
- Section 32.7. rip\_usrreq Function
- Section 32.8. rip\_ctloutput Function
- Section 32.9. Summary

**Epilogue**

**Appendix A. Solutions to Selected Exercises**

- Chapter 1
- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7
- Chapter 8
- Chapter 9
- Chapter 10
- Chapter 11
- Chapter 12
- Chapter 13
- Chapter 14
- Chapter 15
- Chapter 16
- Chapter 17
- Chapter 18
- Chapter 19
- Chapter 20
- Chapter 21
- Chapter 22
- Chapter 23
- Chapter 24

[Chapter 25](#)

[Chapter 26](#)

[Chapter 27](#)

[Chapter 28](#)

[Chapter 29](#)

[Chapter 30](#)

[Chapter 31](#)

[Chapter 32](#)

## Appendix B. Source Code Availability

[URLs: Uniform Resource Locators](#)

[4.4BSD-Lite](#)

[Operating Systems that Run the 4.4BSD-Lite Networking Software](#)

[RFCs](#)

[GNU Software](#)

[PPP Software](#)

[mrouted Software](#)

[ISODE Software](#)

## Appendix C. RFC 1122 Compliance

[Section C.1. Link-Layer Requirements](#)

[Section C.2. IP Requirements](#)

[Section C.3. IP Options Requirements](#)

[Section C.4. IP Fragmentation and Reassembly Requirements](#)

[Section C.5. ICMP Requirements](#)

[Section C.6. Multicasting Requirements](#)

[Section C.7. IGMP Requirements](#)

[Section C.8. Routing Requirements](#)

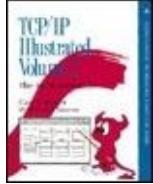
[Section C.9. ARP Requirements](#)

[Section C.10. UDP Requirements](#)

[Section C.11. TCP Requirements](#)

## Bibliography

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed trademarks. Where those designations appear in the book, and we were aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals.

The programs and applications presented in this book have been included for their instructional value. These programs have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:

Pearson Education Corporate Sales Division  
One Lake Street  
Upper Saddle River, NJ 07458  
(800) 382-3419  
[corpsales@pearsontechgroup.com](mailto:corpsales@pearsontechgroup.com)

Visit AW on the Web: [www.awl.com/cseng/](http://www.awl.com/cseng/)

## **Library of Congress Cataloging-in-Publication Data**

(Revised for vol. 2)

Stevens, W. Richard.  
TCP/IP illustrated.

(Addison-Wesley professional computing series)  
Vol. 2 by Gary R. Wright, W. Richard Stevens  
Includes bibliographical references and index  
Contents: v. 1. The protocols v.2. The  
implementation

1. TCP/IP (Computer network protocol) I Wright,  
Gary R., II. Title. III. Series.  
TK5105.55.S74 1994 004.6'2 9:  
ISBN 0-201-63346-9 (v.1)  
ISBN 0-201-63354-X (v.2)

The BSD Daemon used on the cover of this book  
reproduced with the permission of Marshall Kirk  
McKusick.

Copyright © 1995 by Addison-Wesley

All rights reserved. No part of this publication may

reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic or mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

**Text printed on recycled and acid-free paper**

**121314151617 CR 03 02 01 00**

**12th Printing November 2000**

## **Dedication**

*To my parents and my sister,*

*for their love and support.*

*G.R.W.*

*To my parents,*

*for the gift of an education,*

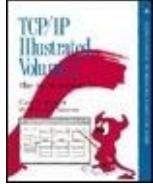
*and the example of a work ethic.*

*W.R.S.*

---

**Team-Fly**





[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Preface

Introduction

Organization of the Book

Intended Audience

Source Code Copyright

Acknowledgments

---

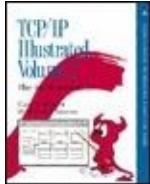
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Preface

# Introduction

This book describes and presents the source code for the common reference implementation of TCP/IP: the implementation from the Computer Systems Research Group (CSRG) at the University of California at Berkeley. Historically this has been distributed with the 4.x BSD system (Berkeley Software Distribution). This implementation was first released in 1982 and has survived many significant changes, much fine tuning, and numerous ports to other Unix and non-Unix systems. This is not a toy implementation, but the foundation for TCP/IP implementations that are run daily on hundreds of thousands of systems

worldwide. This implementation also provides router functionality, letting us show the differences between a host implementation of TCP/IP and a router.

We describe the implementation and present the entire source code for the kernel implementation of TCP/IP, approximately 15,000 lines of C code. The version of the Berkeley code described in this text is the 4.4BSD-Lite release. This code was made publicly available in April 1994, and it contains numerous networking enhancements that were added to the 4.3BSD Tahoe release in 1988, the 4.3BSD Reno release in 1990, and the 4.4BSD release in 1993.

([Appendix B](#) describes how to obtain this source code.) The 4.4BSD release provides the latest TCP/IP features, such as multicasting and long fat pipe support (for high-bandwidth, long-delay paths). [Figure 1.1](#) provides additional details of the various releases of the Berkeley networking code.

This book is intended for anyone wishing to understand how the TCP/IP protocols

are implemented: programmers writing network applications, system administrators responsible for maintaining computer systems and networks utilizing TCP/IP, and any programmer interested in understanding how a large body of nontrivial code fits into a real operating system.

---

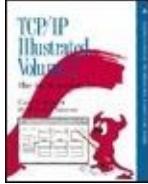
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Preface

# Organization of the Book

The following figure shows the various protocols and subsystems that are covered. The italic numbers by each box indicate the chapters in which that topic is described.

*Chap. 2*

mbufs

7

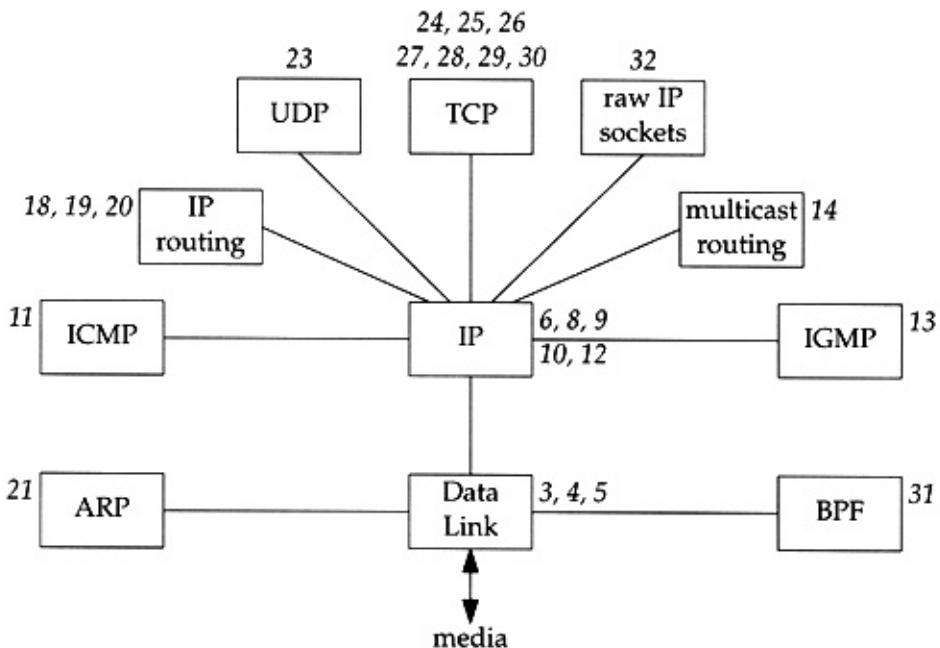
domains

15, 16, 17

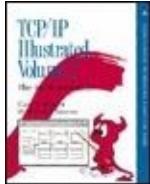
socket  
layer

22

PCBs



We take a bottom-up approach to the TCP/IP protocol suite, starting at the data-link layer, then the network layer (IP, ICMP, IGMP, IP routing, and multicast routing), followed by the socket layer, and finishing with the transport layer (UDP, TCP, and raw IP).



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Preface

# Intended Audience

This book assumes a basic understanding of how the TCP/IP protocols work. Readers unfamiliar with TCP/IP should consult the first volume in this series, [[Stevens 1994](#)], for a thorough description of the TCP/IP protocol suite. This earlier volume is referred to throughout the current text as *Volume 1*. The current text also assumes a basic understanding of operating system principles.

We describe the implementation of the protocols using a data-structures approach. That is, in addition to the source code presentation, each chapter contains pictures and descriptions of the data

structures used and maintained by the source code. We show how these data structures fit into the other data structures used by TCP/IP and the kernel. Heavy use is made of diagrams throughout the textthere are over 250 diagrams.

This data-structures approach allows readers to use the book in various ways. Those interested in all the implementation details can read the entire text from start to finish, following through all the source code. Others might want to understand how the protocols are implemented by understanding all the data structures and reading all the text, but not following through all the source code.

We anticipate that many readers are interested in specific portions of the book and will want to go directly to those chapters. Therefore many forward and backward references are provided throughout the text, along with a thorough index, to allow individual chapters to be studied by themselves. The inside back covers contain an alphabetical cross-reference of all the functions and macros

described in the book and the starting page number of the description. Exercises are provided at the end of the chapters; most solutions are in [Appendix A](#) to maximize the usefulness of the text as a self-study reference.

---

[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)

## Preface

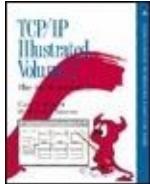
# Source Code Copyright

All of the source code presented in this book, or in the accompanying CD-ROM, is available under a free software license. It is released under the terms of the GNU General Public License (GPL). This software is publicly available for download from the Internet.

All of this source code contains the following copyright notice:

```
/*
 * Copyright (c) 1982, 1986, 1988, 1990, 1991 by
 *          The Regents of the University of California.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
```

\* documentation and/or other mat  
\* 3. All advertising materials ment  
\* must display the following ack  
\*       This product includes softwa  
\*       California, Berkeley and its  
\* 4. Neither the name of the Univer  
\* may be used to endorse or pron  
\* without specific prior writer  
\*  
\* THIS SOFTWARE IS PROVIDED BY THE  
\* ANY EXPRESS OR IMPLIED WARRANTIES  
\* IMPLIED WARRANTIES OF MERCHANTABILITY  
\* ARE DISCLAIMED. IN NO EVENT SHAI  
\* FOR ANY DIRECT, INDIRECT, INCIDEN  
\* DAMAGES (INCLUDING, BUT NOT LIMIT  
\* OR SERVICES; LOSS OF USE, DATA, C  
\* HOWEVER CAUSED AND ON ANY THEORY  
\* LIABILITY, OR TORT (INCLUDING NEG  
\* OUT OF THE USE OF THIS SOFTWARE,  
\* SUCH DAMAGE.  
\*/



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Preface

# Acknowledgments

We thank the technical reviewers who read the manuscript and provided important feedback on a tight timetable: Ragnvald Blindheim, Jon Crowcroft, Sally Floyd, Glen Glater, John Gulbenkian, Don Hering, Mukesh Kacker, Berry Kercheval, Brian W. Kernighan, Ulf Kieber, Mark Laubach, Steven McCanne, Craig Partridge, Vern Paxson, Steve Rago, Chakravardhi Ravi, Peter Salus, Doug Schmidt, Keith Sklower, Ian Lance Taylor, and G. N. Ananda Vardhana. A special thanks to the consulting editor, Brian Kernighan, for his rapid, thorough, and helpful reviews throughout the course of the project, and for his continued encouragement and

support.

Our thanks (again) to the National Optical Astronomy Observatories (NOAO), especially Sidney Wolff, Richard Wolff, and Steve Grandi, for providing access to their networks and hosts. Our thanks also to the U.C. Berkeley CSRG: Keith Bostic and Kirk McKusick provided access to the latest 4.4BSD system, and Keith Sklower provided the modifications to the 4.4BSD-Lite software to run under BSD/386 V1.1.

G.R.W. wishes to thank John Wait, for several years of gentle prodding; Dave Schaller, for his encouragement; and Jim Hogue, for his support during the writing and production of this book.

W.R.S. thanks his family, once again, for enduring another "small" book project. Thank you Sally, Bill, Ellen, and David.

The hardwork, professionalism, and support of the team at Addison-Wesley has made the authors' job that much easier. In particular, we wish to thank John Wait for his guidance and Kim Dawley for her

creative ideas.

Camera-ready copy of the book was produced by the authors. It is only fitting that a book describing an industrial-strength software system be produced with an industrial-strength text processing system. Therefore one of the authors chose to use the Groff package written by James Clark, and the other author agreed begrudgingly.

We welcome electronic mail from any readers with comments, suggestions, or bug fixes: [tcpipiv2-book@aw.com](mailto:tcpipiv2-book@aw.com). Each author will gladly blame the other for any remaining errors.

Gary R. Wright  
<http://www.connix.com/~gwright>  
*Middletown, Connecticut*

W. Richard Stevens  
<http://www.kohala.com/~rstevens>  
*Tucson, Arizona*

*November 1994*

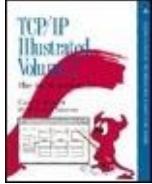
---

**Team-Fly** 

[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 1. Introduction

Section 1.1. Introduction

Section 1.2. Source Code Presentation

Section 1.3. History

Section 1.4. Application Programming  
Interfaces

Section 1.5. Example Program

Section 1.6. System Calls and Library  
Functions

Section 1.7. Network Implementation  
Overview

Section 1.8. Descriptors

Section 1.9. Mbufs (Memory Buffers)  
and Output Processing

Section 1.10. Input Processing

Section 1.11. Network Implementation  
Overview Revisited

[Section 1.12. Interrupt Levels and Concurrency](#)

[Section 1.13. Source Code Organization](#)

[Section 1.14. Test Network](#)

[Section 1.15. Summary](#)

---

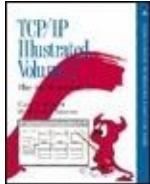
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.1 Introduction

This chapter provides an introduction to the Berkeley networking code. We start with a description of the source code presentation and the various typographical conventions used throughout the text. A quick history of the various releases of the code then lets us see where the source code shown in this book fits in. This is followed by a description of the two predominant programming interfaces used under both Unix and non-Unix systems to write programs that use the TCP/IP protocols.

We then show a simple user program that sends a UDP datagram to the daytime

server on another host on the local area network, causing the server to return a UDP datagram with the current time and date on the server as a string of ASCII text. We follow the datagram sent by the process all the way down the protocol stack to the device driver, and then follow the reply received from server all the way up the protocol stack to the process. This trivial example lets us introduce many of the kernel data structures and concepts that are described in detail in later chapters.

The chapter finishes with a look at the organization of the source code that is presented in the book and a review of where the networking code fits in the overall organization.

## Chapter 1. Introduction

---

### 1.2 Source Code Presentation

Presenting 15,000 lines of source code, regardless of how it is presented, is a challenge. The following format is used for all the source code in this volume.

---

```
381 void
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int      errno;
385 {
386     struct tcpcb *tp = intotcpck
387     if (tp)
388         tp->snd_cwnd = tp->t_max
389 }
```

---

## Set congestion window to one segment

387-388

This is the `tcp_quench` function from the file `tcp.c` files in the 4.4BSD-Lite distribution, which we c line is numbered. The text describing portions ending line numbers in the left margin, as show paragraph is preceded by a short descriptive he the code being described.

The source code has been left as is from the 4. occasional bugs, which we note and discuss wh comments from the original authors. The code program to provide consistency in appearance. column boundaries to allow the lines to fit on a corresponding `#endif` have been removed wh GATEWAY and MROUTING, since we assume th multicast router). All register specifiers have be been added and typographical errors in the con code has been left alone.

The functions vary in size from a few lines (`tcp_` which is the biggest at 1100 lines. Functions th broken into pieces, which are shown one after t

the code and its accompanying description on t  
isn't always possible without wasting a large am

Many cross-references are provided to other fu  
avoid appending both a figure number and a pa  
back covers contain an alphabetical cross-refer  
described in the book, and the starting page nu  
code in the book is taken from the publicly avai  
obtain a copy: [Appendix B](#) details various ways  
copy to search through [e.g., with the Unix gre

Each chapter that describes a source code mod  
source files being described, followed by the glo  
maintained by the code, some sample statistics  
SNMP variables related to the protocol being de  
defined across various source files and headers  
reference. Showing all the statistics at this poi  
code when the statistics are updated. Chapter :  
SNMP. Our interest in this text is in the informa  
the kernel to support an SNMP agent running o

## Typographical Conventions

In the figures throughout the text we use a cor  
the names of structure members (`m_next`), a s  
are defined constants (`NULL`) or constant value  
with braces for structure names (`mbuf{}`). He

mbuf {}	
m_next	NULL
m_len	512

In tables we use a constant-width font for various members, and the slanted constant-width font for an example:

m_flags	
M_BCAST	sent/received as link-level broadcast

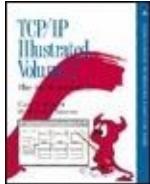
We normally show all #define symbols this way unless necessary (the value of M\_BCAST is irrelevant) unless some other ordering makes sense.

Throughout the text we'll use indented, parenthesized historical points or implementation minutiae.

We refer to Unix commands using the name of parentheses, as in grep(1). The number in parentheses is the 4.4BSD manual of the "manual page" for the command can be located.

**Team-Fly**





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.3 History

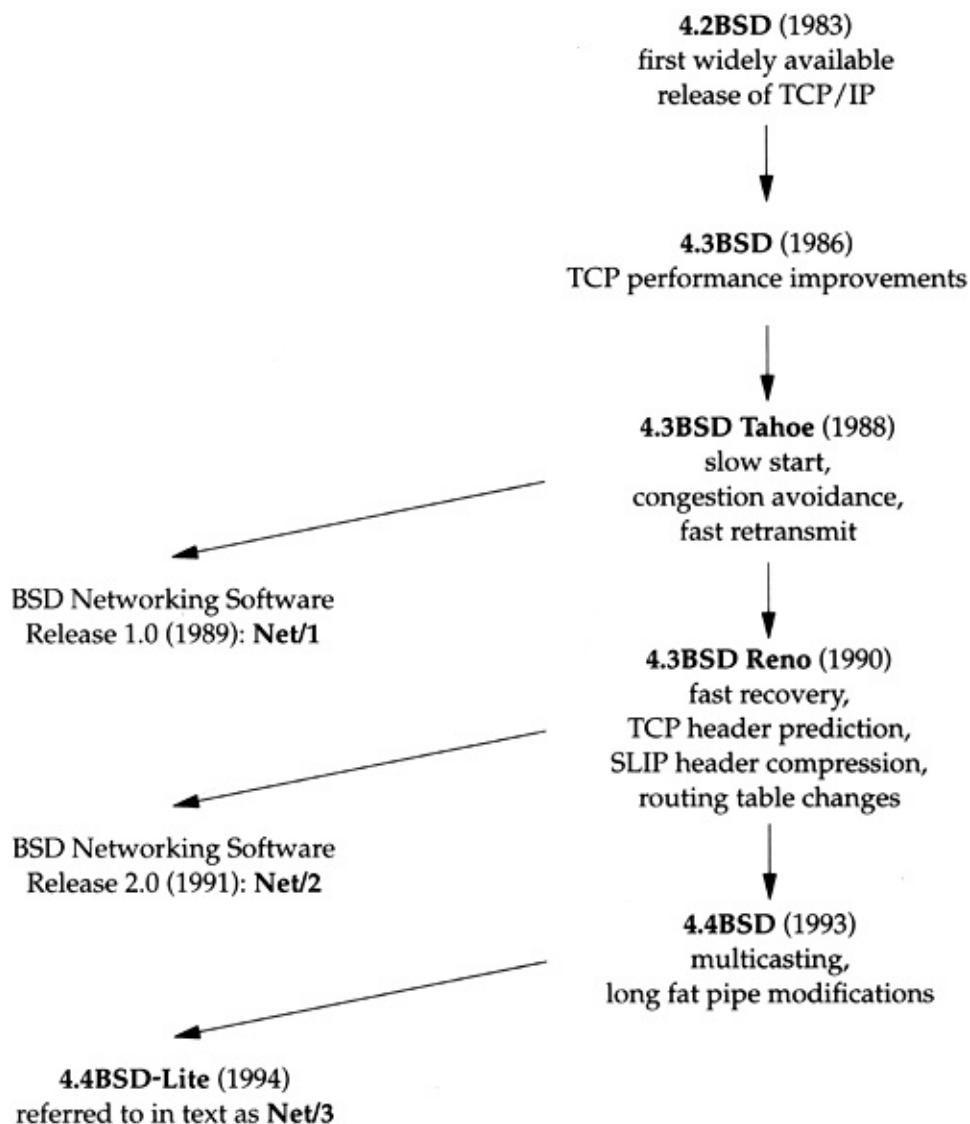
This book describes the common reference implementation of TCP/IP from the Computer Systems Research Group at the University of California at Berkeley.

Historically this has been distributed with the 4.x BSD system (Berkeley Software Distribution) and with the "BSD Networking Releases." This source code has been the starting point for many other implementations, both for Unix and non-Unix operating systems.

[Figure 1.1](#) shows a chronology of the various BSD releases, indicating the important TCP/IP features. The releases shown on the left side are publicly

available source code releases containing all of the networking code: the protocols themselves, the kernel routines for the networking interface, and many of the applications and utilities (such as Telnet and FTP).

**Figure 1.1. Various BSD releases with important TCP/IP features.**



Although the official name of the software described in this text is the *4.4BSD-Lite* distribution, we'll refer to it simply as *Net/3*.

While the source code is distributed by U. C. Berkeley and is called the *Berkeley Software Distribution*, the TCP/IP code is really the merger and consolidation of the

works of various researchers, both at Berkeley and at other locations.

Throughout the text we'll use the term *Berkeley-derived implementation* to refer to vendor implementations such as SunOS 4.x, System V Release 4 (SVR4), and AIX 3.2, whose TCP/IP code was originally developed from the Berkeley sources. These implementations have much in common, often including the same bugs!

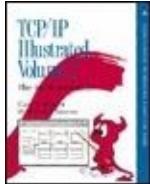
Not shown in [Figure 1.1](#) is that the first release with the Berkeley networking code was actually 4.1cBSD in 1982. 4.2BSD, however, was the widely released version in 1983.

BSD releases prior to 4.1cBSD used a TCP/IP implementation developed at Bolt Beranek and Newman (BBN) by Rob Gurwitz and Jack Haverty. [Chapter 18](#) of [[Salus 1994](#)] provides additional details on the incorporation of the BBN code into 4.2BSD. Another influence on the Berkeley TCP/IP code was the TCP/IP implementation done by Mike Muuss at the Ballistics Research Lab for

the PDP-11.

Limited documentation exists on the changes in the networking code from one release to the next. [[Karels and McKusick 1986](#)] describe the changes from 4.2BSD to 4.3BSD, and [[Jacobson 1990d](#)] describes the changes from 4.3BSD Tahoe to 4.3BSD Reno.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

# 1.4 Application Programming Interfaces

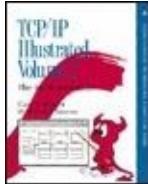
Two popular *application programming interfaces* (APIs) for writing programs to use the Internet protocols are *sockets* and *TLI* (Transport Layer Interface). The former is sometimes called *Berkeley sockets*, since it was widely released with the 4.2BSD system ([Figure 1.1](#)). It has, however, been ported to many non-BSD Unix systems and many non-Unix systems. The latter, originally developed by AT&T, is sometimes called *XTI* (X/Open Transport Interface) in recognition of the work done by X/Open, an international group of computer vendors who produce their own set of standards. XTI is effectively a

superset of TLI.

This is not a programming text, but we describe the sockets interface since sockets are used by applications to access TCP/IP in Net/3 (and in all other BSD releases). The sockets interface has also been implemented on a wide variety of non-Unix systems. The programming details for both sockets and TLI are available in [[Stevens 1990](#)].

System V Release 4 (SVR4) also provides a sockets API for applications to use, although the implementation differs from what we present in this text. Sockets in SVR4 are based on the "streams" subsystem that is described in [[Rago 1993](#)].





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

## 1.5 Example Program

We'll use the simple C program shown in [Figure 1.2](#) to introduce many features of the BSD networking implementation in this chapter.

**Figure 1.2. Example program: send a datagram to the UDP daytime server and read a response.**

---

```
1 /*  
2  * Send a UDP datagram to the daytime server on some other host.  
3  * read the reply, and print the time and date on the server.  
4 */  
5 #include <sys/types.h>  
6 #include <sys/socket.h>  
7 #include <netinet/in.h>  
8 #include <arpa/inet.h>  
9 #include <stdio.h>  
10 #include <stdlib.h>  
11 #include <string.h>  
12 #define BUFFSIZE 150      /* arbitrary size */
```

```
13 int
14 main()
15 {
16     struct sockaddr_in serv;
17     char    buff[BUFFSIZE];
18     int      sockfd, n;
19     if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
20         err_sys("socket error");
21     bzero((char *) &serv, sizeof(serv));
22     serv.sin_family = AF_INET;
23     serv.sin_addr.s_addr = inet_addr("140.252.1.32");
24     serv.sin_port = htons(13);
25     if (sendto(sockfd, buff, BUFFSIZE, 0,
26                (struct sockaddr *) &serv, sizeof(serv)) != BUFFSIZE)
27         err_sys("sendto error");
28     if ((n = recvfrom(sockfd, buff, BUFFSIZE, 0,
29                        (struct sockaddr *) NULL, (int *) NULL)) < 2)
30         err_sys("recvfrom error");
31     buff[n - 2] = 0; /* null terminate */
32     printf("%s\n", buff);
33     exit(0);
34 }
```

---

## Create a datagram socket

19-20

socket creates a UDP socket and returns a descriptor to the process, which is stored in the variable sockfd. The error-handling function err\_sys is shown in Appendix B.2 of [Stevens 1992]. It accepts any number of arguments, formats them using vsprintf, prints the Unix error message corresponding to the errno value from the system call, and then terminates the process.

We've now used the term *socket* in

three different ways. (1) The API developed for 4.2BSD to allow programs to access the networking protocols is normally called the *sockets API* or just the *sockets interface*. (2) socket is the name of a function in the sockets API. (3) We refer to the end point created by the call to socket as a socket, as in the comment "create a datagram socket."

Unfortunately, there are still more uses of the term *socket*. (4) The return value from the socket function is called a *socket descriptor* or just a *socket*. (5) The Berkeley implementation of the networking protocols within the kernel is called the *sockets implementation*, compared to the System V streams implementation, for example. (6) The combination of an IP address and a port number is often called a *socket*, and a pair of IP addresses and port numbers is called a *socket pair*. Fortunately, it is usually obvious from the discussion what the term *socket* refers to.

## Fill in sockaddr\_in structure with server's address

21-24

An Internet socket address structure (sockaddr\_in) is filled in with the IP address (140.252.1.32) and port number (13) of the daytime server. Port number 13 is the standard Internet daytime server, provided by most TCP/IP implementations [[Stevens 1994](#), Fig. 1.9]. Our choice of the server host is arbitrarywe just picked a local host ([Figure 1.17](#)) that provides the service.

The function inet\_addr takes an ASCII character string representing a *dotted-decimal* IP address and converts it into a 32-bit binary integer in the network byte order. (The network byte order for the Internet protocol suite is big endian. [[Stevens 1990](#), Chap. 4] discusses host and network byte order, and little versus big endian.) The function htons takes a short integer in the host byte order (which could be little endian or big endian) and

converts it into the network byte order (big endian). On a system such as a Sparc, which uses big endian format for integers, htons is typically a macro that does nothing. In BSD/386, however, on the little endian 80386, htons can be either a macro or a function that swaps the 2 bytes in a 16-bit integer.

## Send datagram to server

25-27

The program then calls sendto, which sends a 150-byte datagram to the server. The contents of the 150-byte buffer are indeterminate since it is an uninitialized array allocated on the run-time stack, but that's OK for this example because the server never looks at the contents of the datagram that it receives. When the server receives a datagram it sends a reply to the client. The reply contains the current time and date on the server in a human-readable format.

Our choice of 150 bytes for the client's

datagram is arbitrary. We purposely pick a value greater than 100 and less than 208 to show the use of an mbuf chain later in this chapter. We also want a value less than 1472 to avoid fragmentation on an Ethernet.

## Read datagram returned by server

28-32

The program reads the datagram that the server sends back by calling recvfrom. Unix servers typically send back a 26-byte string of the form

Sat Dec 11 11:28:05 1993\r\n

where \r is an ASCII carriage return and \n is an ASCII linefeed. Our program overwrites the carriage return with a null byte and calls printf to output the result.

We go into lots of detail about various parts of this example in this and later chapters as we examine the implementation of the functions socket,

sendto, and recvfrom.

---

**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)

## Chapter 1. Introduction

### 1.6 System Calls and Library Functions

All operating systems provide service points through which applications communicate with the kernel. All variants of Unix provide a well-defined interface to the kernel, known as *system calls*. We cannot change the system call interface without changing the kernel's code. Unix Version 7 provided about 50 system calls; the current version of Solaris has around 120.

The system call interface is documented in Section 1.5. The definition of each system call is in the C language, regardless of how it is implemented by the particular system.

The Unix technique is for each system call to have a corresponding function in the standard C library. An application calls this function, passing the arguments required by the system call. This function then invokes the appropriate kernel routine, which performs the requested operation on the system. For example, the function may save the state of all general registers and then execute some machine code to cause a hardware interrupt into the kernel. For our purposes, we

Section 3 of the *Unix Programmer's Manual* defines several functions for programmers. These functions are not entry points for one or more of the kernel's system calls. For example, the write system call to perform the output, but the functions that convert ASCII to integer) don't involve the operating system.

From an implementor's point of view, the distinction between a function and a system call is fundamental. From a user's perspective, however, the distinction is irrelevant. For example, if we run [Figure 1.2](#) under 4.4BSD, we see that socket, sendto, and recvfrom, each ends up calling the kernel. We show the BSD kernel implementation below.

If we run the program under SVR4, where the socket function is implemented by the "streams" subsystem, the interaction of the different. Under SVR4 the call to socket ends up being pushed into a file /dev/udp and then pushes the streams module. The call to sendto results in a putmsg system call, and the call to recvfrom results in a getmsg system call. These SVR4 details are not critical to our discussion, but the implementation can be totally different while providing the same interface.

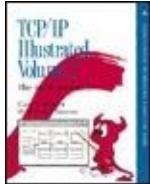
This difference in implementation technique also applies to the sendto function appearing in Section 2 of the 4.4BSD manual (in the networking subsection of Section 3) of the manual.

Finally, the implementation technique can change from version to version. In Net/1 send and sendto were implemented as library functions, but in Net/3, however, send is a library function that calls the kernel's system call.

```
send(int s, char *msg, int len,
{
    return(sendto(s, msg, len, f
}
```

The advantage in implementing send as a library function is in the number of system calls and in the amount of code. The disadvantage is the additional overhead of one more function call.

Since this text describes the Berkeley implementation, the calls made by the process (socket, bind, connect, etc.) are system calls.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

# 1.7 Network Implementation Overview

Net/3 provides a general purpose infrastructure capable of simultaneously supporting multiple communication protocols. Indeed, 4.4BSD supports four distinct communication protocol families:

### **1. TCP/IP (the Internet protocol suite), the topic of this book.**

- XNS (Xerox Network Systems), a protocol suite that is similar to TCP/IP; it was popular in the mid-1980s for connecting Xerox hardware (such as printers and file servers), often using an Ethernet. Although the code is still

distributed with Net/3, few people use this protocol suite today, and many vendors who use the Berkeley TCP/IP code remove the XNS code (so they don't have to support it).

- The OSI protocols [[Rose 1990](#); [Piscitello and Chapin 1993](#)]. These protocols were designed during the 1980s as the ultimate in open-systems technology, to replace all other communication protocols. Their appeal waned during the early 1990s, and as of this writing their use in real networks is minimal. Their place in history is still to be determined.
- The Unix domain protocols. These do not form a true protocol suite in the sense of communication protocols used to exchange information between different systems, but are provided as a form of *interprocess communication* (IPC).

The advantage in using the Unix domain protocols for IPC between two processes on the same host, versus other forms of IPC such as System V message queues [[Stevens 1990](#)], is that the Unix domain

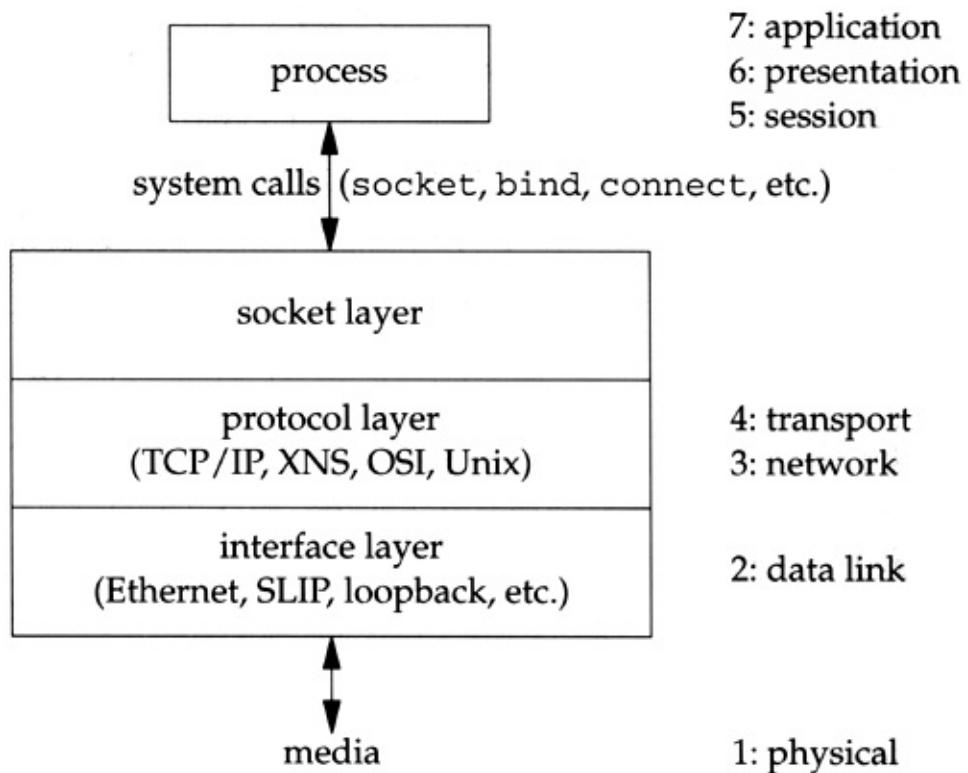
protocols are accessed using the same API (sockets) as are the other three communication protocols. Message queues, on the other hand, and most other forms of IPC, have an API that is completely different from both sockets and TLI. Having IPC between two processes on the same host use the networking API makes it easy to migrate a client-server application from one host to many hosts. Two different protocols are provided in the Unix domain: a reliable, connection-oriented, byte-stream protocol that looks like TCP, and an unreliable, connectionless, datagram protocol that looks like UDP.

Although the Unix domain protocols can be used as a form of IPC between two processes on the same host, these processes could also use TCP/IP to communicate with each other. There is no requirement that processes communicating using the Internet protocols reside on different hosts.

The networking code in the kernel is organized into three layers, as shown in [Figure 1.3](#). On the right side of this figure

we note where the seven layers of the OSI reference model [Piscitello and Chapin 1993] fit in the BSD organization.

**Figure 1.3. The general organization of networking code in Net/3.**

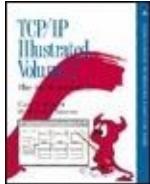


1. **The *socket layer* is a protocol-independent interface to the protocol-dependent layer below. All system calls start at the protocol-independent socket layer. For example, the protocol-independent**

**code in the socket layer for the bind system call comprises a few dozen lines of code: these verify that the first argument is a valid socket descriptor and that the second argument is a valid pointer in the process. The protocol-dependent code in the layer below is then called, which might comprise hundreds of lines of code.**

- The *protocol layer* contains the implementation of the four protocol families that we mentioned earlier (TCP/IP, XNS, OSI, and Unix domain). Each protocol suite may have its own internal structure, which we don't show in [Figure 1.3](#). For example, in the Internet protocol suite, IP is the lowest layer (the network layer) with the two transport layers (TCP and UDP) above IP.
- The *interface layer* contains the device drivers that communicate with the network devices.

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.8 Descriptors

Figure 1.2 begins with a call to `socket`, specifying the type of socket desired. The combination of the Internet protocol family (`PF_INET`) and a datagram socket (`SOCK_DGRAM`) gives a socket whose protocol is UDP.

The return value from `socket` is a descriptor that shares all the properties of other Unix descriptors: read and write can be called for the descriptor, you can dup it, it is shared by the parent and child after a call to `fork`, its properties can be modified by calling `fcntl`, it can be closed by calling `close`, and so on. We see in our example that the socket descriptor is the first

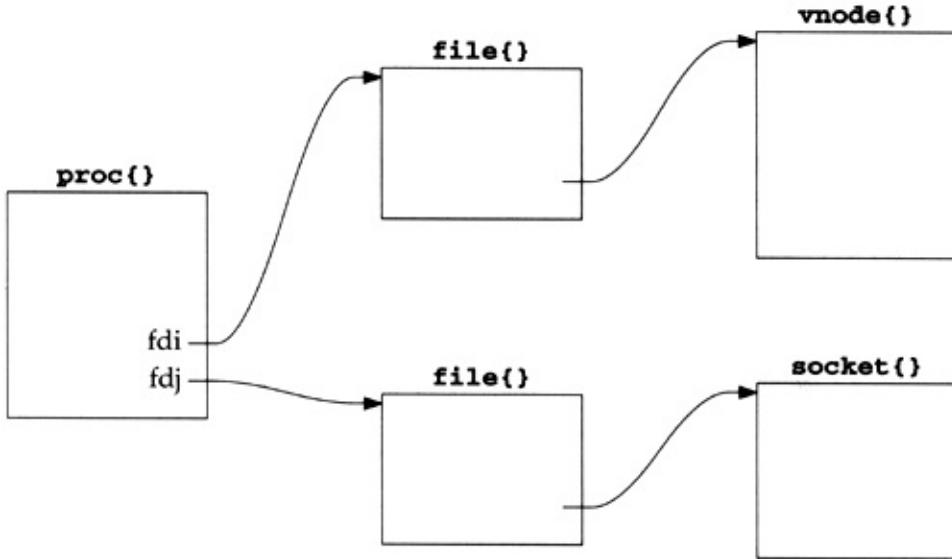
argument to both the sendto and recvfrom functions. When our program terminates (by calling exit), all open descriptors including the socket descriptor are closed by the kernel.

We now introduce the data structures that are created by the kernel when the process calls socket. We describe these data structures in more detail in later chapters.

Everything starts with the process table entry for the process. One of these exists for each process during its lifetime.

A descriptor is an index into an array within the process table entry for the process. This array entry points to an open file table structure, which in turn points to an i-node or v-node structure that describes the file. [Figure 1.4](#) summarizes this relationship.

**Figure 1.4. Fundamental relationship between kernel data structures starting with a descriptor.**



In this figure we also show a descriptor that refers to a socket, which is the focus of this text. We place the notation `proc{}` above the process table entry, since its definition in C is

```

struct proc {
    ...
}

```

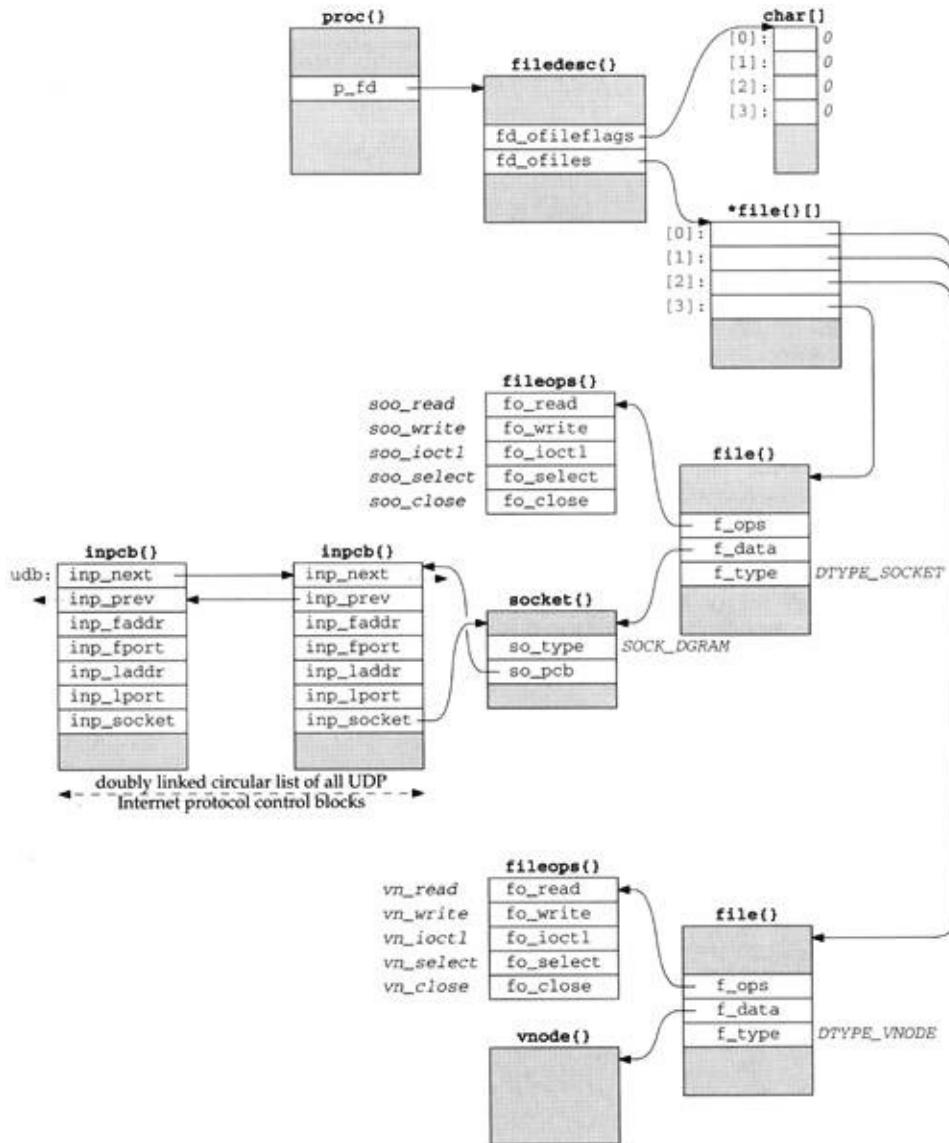
and we use this notation for structures in our figures throughout the text.

[[Stevens 1992](#), Sec. 3.10] shows how the relationships between the descriptor, file table structure, and i-node or v-node change as the process calls `dup` and `fork`. The relationships between these three

data structures exists in all versions of Unix, although the details change with different implementations. Our interest in this text is with the socket structure and the Internet-specific data structures that it points to. But we need to understand how a descriptor leads to a socket structure, since the socket system calls start with a descriptor.

[Figure 1.5](#) shows more details of the Net/3 data structures for our example program, if the program is executed as

**Figure 1.5. Kernel data structures after call to socket in example program.**



without redirecting standard input (descriptor 0), standard output (descriptor 1), or standard error (descriptor 2). In this example, descriptors 0, 1, and 2 are connected to our terminal, and the lowest-numbered unused descriptor is 3 when

socket is called.

When a process executes a system call such as socket, the kernel has access to the process table structure. The entry p\_fd in this structure points to the filedesc structure for the process. There are two members of this structure that interest us now: fd\_ofileflags is a pointer to an array of characters (the per-descriptor flags for each descriptor), and fd\_ofiles is a pointer to an array of pointers to file table structures. The per-descriptor flags are 8 bits wide since only 2 bits can be set for any descriptor: the close-on-exec flag and the mapped-from-device flag. We show all these flags as 0.

We purposely call this section "[Descriptors](#)" and not "File Descriptors" since Unix descriptors can refer to lots of things other than files: sockets, pipes, directories, devices, and so on. Nevertheless, much of Unix literature uses the adjective *file* when talking about descriptors, which is an unnecessary qualification. Here the kernel data structure is called filedesc{}

even though we're about to describe socket descriptors. We'll use the unqualified term *descriptor* whenever possible.

The data structure pointed to by the `fd_ofiles` entry is shown as `*file{}[]` since it is an array of pointers to file structures. The index into this array and the array of descriptor flags is the nonnegative descriptor itself: 0, 1, 2, and so on. In [Figure 1.5](#) we show the entries for descriptors 0, 1, and 2 pointing to the same file structure at the bottom of the figure (since all three descriptors refer to our terminal). The entry for descriptor 3 points to a different file structure for our socket descriptor.

The `f_type` member of the file structure specifies the descriptor type as either `DTYPE_SOCKET` or `DTYPE_VNODE`. V-nodes are a general mechanism that allows the kernel to support different types of filesystems: a disk filesystem, a network filesystem (such as NFS), a filesystem on a CD-ROM, a memory-based filesystem, and so on. Our interest in this text is not with

v-nodes, since TCP/IP sockets always have a type of DTTYPE\_SOCKET.

The f\_data member of the file structure points to either a socket structure or a vnode structure, depending on the type of descriptor. The f\_ops member points to a vector of five function pointers. These function pointers are used by the read, readv, write, writev, ioctl, select, and close system calls, since these system calls work with either a socket descriptor or a nonsocket descriptor. Rather than look at the f\_type value each time one of these system calls is invoked and then jump accordingly, the implementors chose always to jump indirectly through the corresponding entry in the fileops structure instead.

Notationally we use a fixed-width font (`fo_read`) to show the name of a structure member and a slanted fixed-width font (`soo_read`) to show the contents of a structure member. Also note that sometimes we show the pointer to a structure arriving at the top left corner (e.g., the filedesc structure) and

sometimes at the top right corner (e.g., both file structures and both fileops structures). This is to simplify the figures.

Next we come to the socket structure that is pointed to by the file structure when the descriptor type is DTYPE\_SOCKET. In our example, the socket type (SOCK\_DGRAM for a datagram socket) is stored in the so\_type member. An Internet protocol control block (PCB) is also allocated: an inpcb structure. The so\_pcb member of the socket structure points to the inpcb, and the inp\_socket member of the inpcb structure points to the socket structure. Each points to the other because the activity for a given socket can occur from two directions: "above" or "below."

- 1. When the process executes a system call, such as sendto, the kernel starts with the descriptor value and uses fd\_ofiles to index into the vector of file structure pointers, ending up with the file structure for the descriptor. The file structure points to the socket structure, which points to the inpcb**

## **structure.**

- When a UDP datagram arrives on a network interface, the kernel searches through all the UDP protocol control blocks to find the appropriate one, minimally based on the destination UDP port number and perhaps the destination IP address, source IP address, and source port numbers too. Once the inpcb structure is located, the kernel finds the corresponding socket structure through the `inp_socket` pointer.

The members `inp_faddr` and `inp_laddr` contain the foreign and local IP addresses, and the members `inp_fport` and `inp_lport` contain the foreign and local port numbers. The combination of the local IP address and the local port number is often called a *socket*, as is the combination of the foreign IP address and the foreign port number.

We show another inpcb structure with the name `udb` on the left in [Figure 1.5](#). This is a global structure that is the head of a linked list of all UDP PCBs. We show the

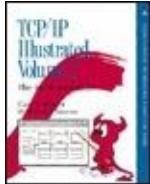
two members `inp_next` and `inp_prev` that form a doubly linked circular list of all UDP PCBs. For notational simplicity in the figure, we show two parallel horizontal arrows for the two links instead of trying to have the heads of the arrows going to the top corners of the PCBs. The `inp_prev` member of the `inpcb` structure on the right points to the `udb` structure, not the `inp_prev` member of that structure. The dotted arrows from `udb.inp_prev` and the `inp_next` member of the other PCB indicate that there may be other PCBs on the doubly linked list that we don't show.

We've looked at many kernel data structures in this section, most of which are described further in later chapters. The key points to understand now are:

- 1. The call to socket by our process ends up allocating the lowest unused descriptor (3 in our example). This descriptor is used by the process in all subsequent system calls that refer to this socket.**

- The following kernel structures are allocated and linked together: a file structure of type `DTYPE_SOCKET`, a socket structure, and an `inpcb` structure. Lots of initialization is performed on these structures that we don't show: the file structure is marked for read and write (since the call to `socket` always returns a descriptor that can be read or written), the default sizes of the input and output buffers are set in the socket structure, and so on.
- We showed nonsocket descriptors for our standard input, output, and error to show that *all* descriptors end up at a file structure, and it is from that point on that differences appear between socket descriptors and other descriptors.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

# 1.9 Mbufs (Memory Buffers) and Output Processing

A fundamental concept in the design of the Berkeley networking code is the memory buffer, called an *mbuf*, used throughout the networking code to hold various pieces of information. Our simple example ([Figure 1.2](#)) lets us examine some typical uses of mbufs. In [Chapter 2](#) we describe mbufs in more detail.

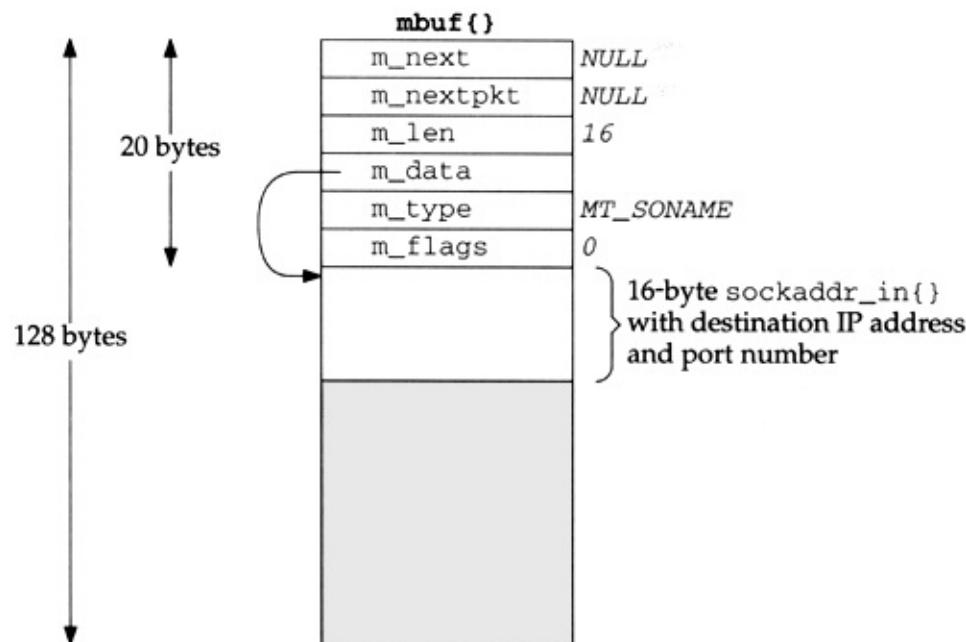
## Mbuf Containing Socket Address Structure

In the call to `sendto`, the fifth argument points to an Internet socket address

structure (named serv) and the sixth argument specifies its length (which we'll see later is 16 bytes). One of the first things done by the socket layer for this system call is to verify that these arguments are valid (i.e., the pointer points to a piece of memory in the address space of the process) and then copy the socket address structure into an mbuf.

Figure 1.6 shows the resulting mbuf.

**Figure 1.6. Mbuf containing destination address for sendto.**



The first 20 bytes of the mbuf is a header

containing information about the mbuf. This 20-byte header contains four 4-byte fields and two 2-byte fields. The total size of the mbuf is 128 bytes.

Mbufs can be linked together using the `m_next` and `m_nexpkt` members, as we'll see shortly. Both are null pointers in this example, which is a stand-alone mbuf.

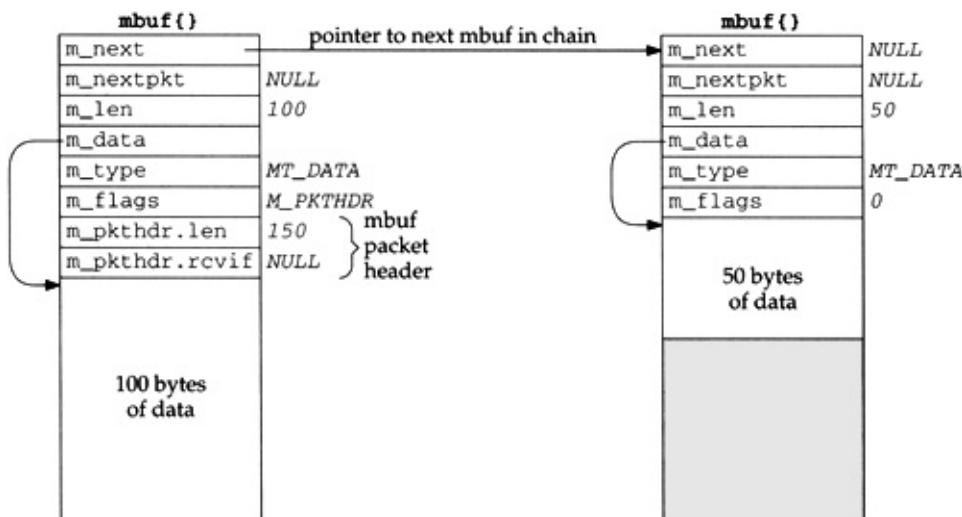
The `m_data` member points to the data in the mbuf and the `m_len` member specifies its length. For this example, `m_data` points to the first byte of data in the mbuf (the byte immediately following the mbuf header). The final 92 bytes of the mbuf data area (108-16) are unused (the shaded portion of [Figure 1.6](#)).

The `m_type` member specifies the type of data contained in the mbuf, which for this example is `MT SONAME` (socket name). The final member in the header, `m_flags`, is zero in this example.

## Mbuf Containing Data

Continuing our example, the socket layer copies the data buffer specified in the call to sendto into one or more mbufs. The second argument to sendto specifies the start of the data buffer (buff), and the third argument is its size in bytes (150). [Figure 1.7](#) shows how two mbufs hold the 150 bytes of data.

**Figure 1.7. Two mbufs holding 150 bytes of data.**



This arrangement is called an *mbuf chain*. The `m_next` member in each mbuf links together all the mbufs in a chain.

The next change we see is the addition of

two members, `m_pkthdr.len` and `m_pkthdr.rcvif`, to the mbuf header in the first mbuf of the chain. These two members comprise the *packet header* and are used only in the first mbuf of a chain. The `m_flags` member contains the value `M_PKTHDR` to indicate that this mbuf contains a packet header. The `len` member of the packet header structure contains the total length of the mbuf chain (150 in this example), and the next member, `rcvif`, we'll see later contains a pointer to the received interface structure for received packets.

Since mbufs are *always* 128 bytes, providing 100 bytes of data storage in the first mbuf on the chain and 108 bytes of storage in all subsequent mbufs on the chain, two mbufs are needed to store 150 bytes of data. We'll see later that when the amount of data exceeds 208 bytes, instead of using three or more mbufs, a different technique is used a larger buffer, typically 1024 or 2048 bytes, called a *cluster* is used.

One reason for maintaining a packet

header with the total length in the first mbuf on the chain is to avoid having to go through all the mbufs on the chain to sum their `m_len` members when the total length is needed.

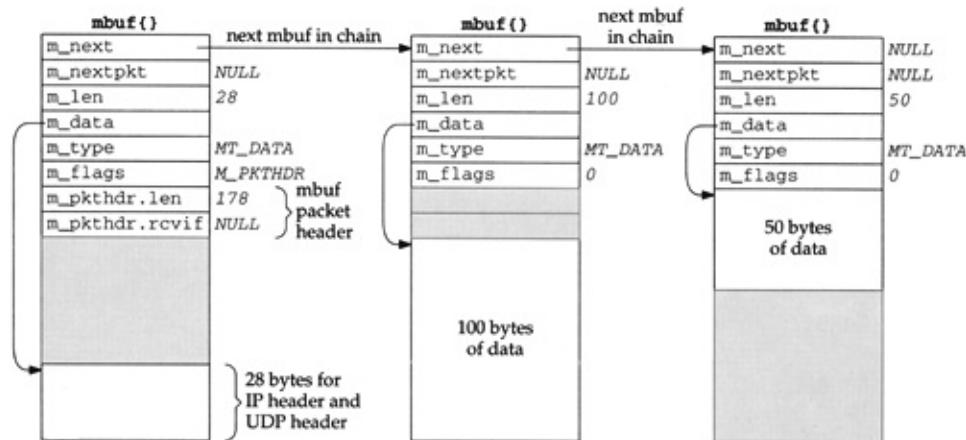
## Prepending IP and UDP Headers

After the socket layer copies the destination socket address structure into an mbuf ([Figure 1.6](#)) and the data into an mbuf chain ([Figure 1.7](#)), the protocol layer corresponding to the socket descriptor (a UDP socket) is called. Specifically, the UDP output routine is called and pointers to the mbufs that we've examined are passed as arguments. This routine needs to prepend an IP header and a UDP header in front of the 150 bytes of data, fill in the headers, and pass the mbufs to the IP output routine.

The way that data is prepended to the mbuf chain in [Figure 1.7](#) is to allocate another mbuf, make it the front of the chain, and copy the packet header from the mbuf with 100 bytes of data into the

new mbuf. This gives us the three mbufs shown in [Figure 1.8](#).

**Figure 1.8. Mbuf chain from Figure 1.7 with another mbuf for IP and UDP headers prepended.**



The IP header and UDP header are stored at the end of the new mbuf that becomes the head of the chain. This allows for any lower-layer protocols (e.g., the interface layer) to prepend its headers in front of the IP header if necessary, without having to copy the IP and UDP headers. The `m_data` pointer in the first mbuf points to the start of these two headers, and `m_len` is 28. Future headers that fit in the 72 bytes of unused space between the packet

header and the IP header can be prepended before the IP header by adjusting the `m_data` pointer and the `m_len` accordingly. Shortly we'll see that the Ethernet header is built here in this fashion.

Notice that the packet header has been moved from the mbuf with 100 bytes of data into the new mbuf. The packet header must always be in the first mbuf on the chain. To accommodate this movement of the packet header, the `M_PKTHDR` flag is set in the first mbuf and cleared in the second mbuf. The space previously occupied by the packet header in the second mbuf is now unused. Finally, the length member in the packet header is incremented by 28 bytes to become 178.

The UDP output routine then fills in the UDP header and as much of the IP header as it can. For example, the destination address in the IP header can be set, but the IP checksum will be left for the IP output routine to calculate and store.

The UDP checksum is calculated and

stored in the UDP header. Notice that this requires a complete pass of the 150 bytes of data stored in the mbuf chain. So far the kernel has made two complete passes of the 150 bytes of user data: once to copy the data from the user's buffer into the kernel's mbufs, and now to calculate the UDP checksum. Extra passes over the data can degrade the protocol's performance, and in later chapters we describe alternative implementation techniques that avoid unnecessary passes.

At this point the UDP output routine calls the IP output routine, passing a pointer to the mbuf chain for IP to output.

## IP Output

The IP output routine fills in the remaining fields in the IP header including the IP checksum, determines the outgoing interface to which the datagram should be given (this is the IP routing function), fragments the IP datagram if necessary, and calls the interface output function.

Assuming the outgoing interface is an Ethernet, a general-purpose Ethernet output function is called, again with a pointer to the mbuf chain as an argument.

## Ethernet Output

The first function of the Ethernet output function is to convert the 32-bit IP address into its corresponding 48-bit Ethernet address. This is done using ARP (Address Resolution Protocol) and may involve sending an ARP request on the Ethernet and waiting for an ARP reply. While this takes place, the mbuf chain to be output is held, waiting for the reply.

The Ethernet output routine then prepends a 14-byte Ethernet header to the first mbuf in the chain, immediately before the IP header ([Figure 1.8](#)). This contains the 6-byte Ethernet destination address, 6-byte Ethernet source address, and 2-byte Ethernet frame type.

The mbuf chain is then added to the end of the output queue for the interface. If the

interface is not currently busy, the interface's "start output" routine is called directly. If the interface is busy, its output routine will process the new mbuf on its queue when it is finished with the buffers already on its output queue.

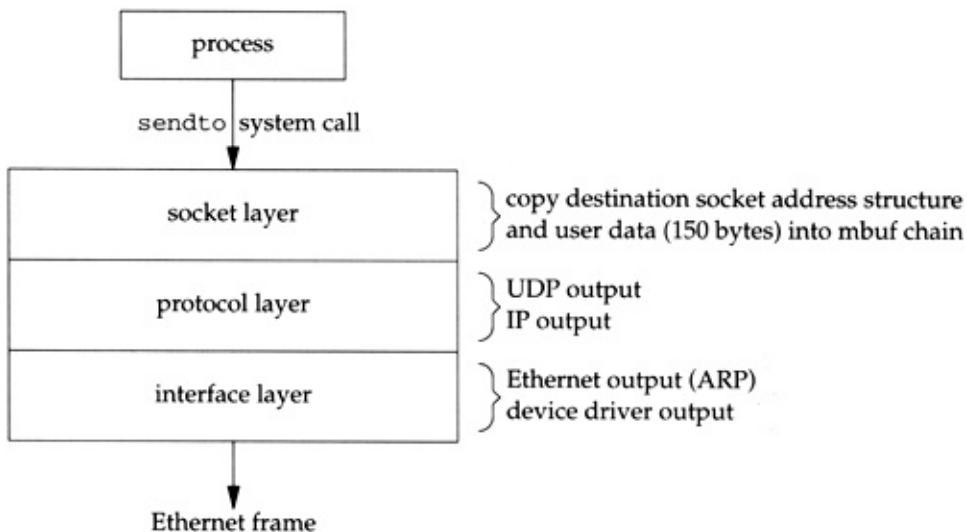
When the interface processes an mbuf that's on its output queue, it copies the data to its transmit buffer and initiates the output. In our example, 192 bytes are copied to the transmit buffer: the 14-byte Ethernet header, 20-byte IP header, 8-byte UDP header, and 150 bytes of user data. This is the third complete pass of the data by the kernel. Once the data is copied from the mbuf chain into the device's transmit buffer, the mbuf chain is released by the Ethernet device driver. The three mbufs are put back into the kernel's pool of free mbufs.

## Summary of UDP Output

In [Figure 1.9](#) we give an overview of the processing that takes place when a process calls `sendto` to transmit a single

UDP datagram. The relationship of the processing that we've described to the three layers of kernel code (Figure 1.3) is also shown.

**Figure 1.9. Processing performed by the three layers for simple UDP output.**



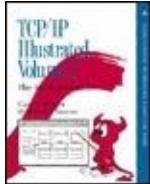
Function calls pass control from the socket layer to the UDP output routine, to the IP output routine, and then to the Ethernet output routine. Each function call passes a pointer to the mbuf chain to be output. At the lowest layer, the device driver, the mbuf chain is placed on the device's output queue and the device is started, if necessary. The function calls return in

reverse order of their call, and eventually the system call returns to the process. Notice that there is no queueing of the UDP data until it arrives at the device driver. The higher layers just prepend their header and pass the mbuf to the next lower layer.

At this point our program calls recvfrom to read the server's reply. Since the input queue for the specified socket is empty (assuming the reply has not been received yet), the process is put to sleep.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.10 Input Processing

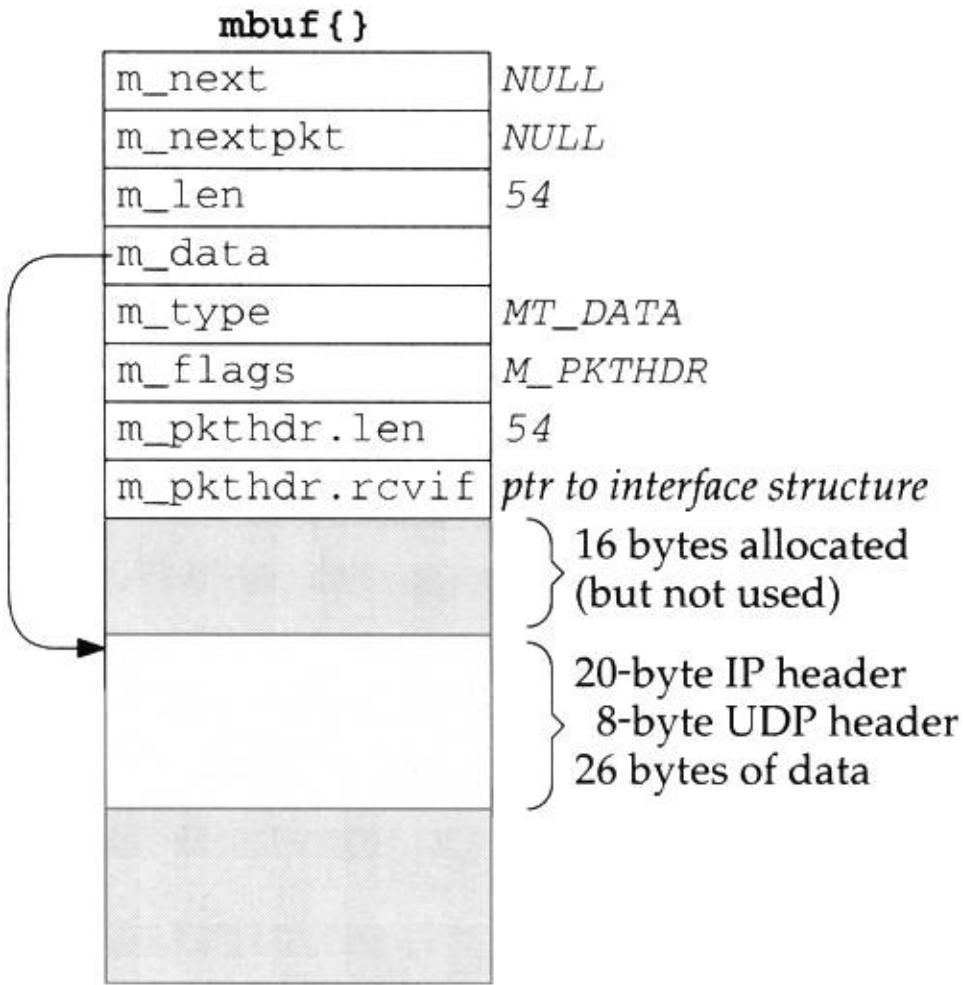
Input processing is different from the output processing just described because the input is *asynchronous*. That is, the reception of an input packet is triggered by a receive-complete interrupt to the Ethernet device driver, not by a system call issued by the process. The kernel handles this device interrupt and schedules the device driver to run.

#### Ethernet Input

The Ethernet device driver processes the interrupt and, assuming it signifies a normal receive-complete condition, the

data bytes are read from the device into an mbuf chain. In our example, 54 bytes of data are received and copied into a single mbuf: the 20-byte IP header, 8-byte UDP header, and 26 bytes of data (the time and date on the server). [Figure 1.10](#) shows the format of this mbuf.

### **Figure 1.10. Single mbuf to hold input Ethernet data.**



This mbuf is a packet header (the M\_PKTHDR flag is set in m\_flags) since it is the first mbuf of a data record. The len member in the packet header contains the total length of data and the recvif. member contains a pointer to the interface structure corresponding to the received interface ([Chapter 3](#)). We see that the recvif member is used for received packets but not for output packets ([Figures 1.7](#) and [1.8](#)).

The first 16 bytes of the data portion of the mbuf are allocated for an interface layer header, but are not used. Since the amount of data (54 bytes) fits in the remaining 84 bytes of the mbuf, the data is stored in the mbuf itself.

The device driver passes the mbuf to a general Ethernet input routine which looks at the type field in the Ethernet frame to determine which protocol layer should receive the packet. In this example, the type field will specify an IP datagram, causing the mbuf to be added to the IP input queue. Additionally, a software interrupt is scheduled to cause the IP input process routine to be executed. The device's interrupt handling is then complete.

## IP Input

IP input is asynchronous and is scheduled to run by a software interrupt. The software interrupt is set by the interface layer when it receives an IP datagram on one of the system's interfaces. When the

IP input routine executes it loops, processing each IP datagram on its input queue and returning when the entire queue has been processed.

The IP input routine processes each IP datagram that it receives. It verifies the IP header checksum, processes any IP options, verifies that the datagram was delivered to the right host (by comparing the destination IP address of the datagram with the host's IP addresses), and forwards the datagram if the system was configured as a router and the datagram is destined for some other IP address. If the IP datagram has reached its final destination, the protocol field in the IP header specifies which protocol's input routine is called: ICMP, IGMP, TCP, or UDP. In our example, the UDP input routine is called to process the UDP datagram.

## UDP Input

The UDP input routine verifies the fields in the UDP header (the length and optional checksum) and then determines whether

or not a process should receive the datagram. In [Chapter 23](#) we discuss exactly how this test is made. A process can receive all datagrams destined to a specified UDP port, or the process can tell the kernel to restrict the datagrams it receives based on the source and destination IP addresses and source and destination port numbers.

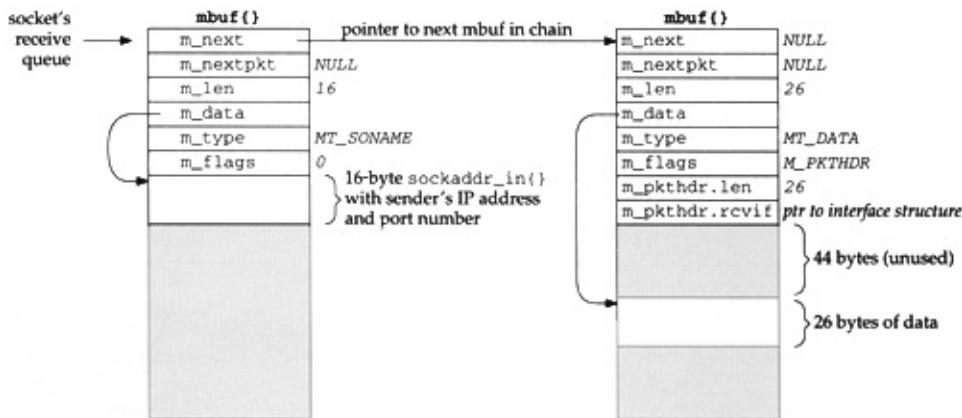
In our example, the UDP input routine starts at the global variable `udb` ([Figure 1.5](#)) and goes through the linked list of UDP protocol control blocks, looking for one with a local port number (`inp_lport`) that matches the destination port number of the received UDP datagram. This will be the PCB created by our call to `socket`, and the `inp_socket` member of this PCB points to the corresponding socket structure, allowing the received data to be queued for the correct socket.

In our example program we never specify the local port number for our application. We'll see in [Exercise 23.3](#) that a side effect of writing the first UDP datagram to a socket that has not

yet bound a local port number is the automatic assignment by the kernel of a local port number (termed an *ephemeral port*) to that socket. That's how the `inp_lport` member of the PCB for our socket gets set to some nonzero value.

Since this UDP datagram is to be delivered to our process, the sender's IP address and UDP port number are placed into an mbuf, and this mbuf and the data (26 bytes in our example) are appended to the receive queue for the socket. [Figure 1.11](#) shows the two mbufs that are appended to the socket's receive queue.

**Figure 1.11. Sender's address and data.**



Comparing the second mbuf on this chain (the one of type MT\_DATA) with the mbuf in [Figure 1.10](#), the m\_len and m\_pkthdr.len members have both been decremented by 28 (20 bytes for the IP header and 8 for the UDP header) and the m\_data pointer has been incremented by 28. This effectively removes the IP and UDP headers, leaving only the 26 bytes of data to be appended to the socket's receive queue.

The first mbuf in the chain contains a 16-byte Internet socket address structure with the sender's IP address and UDP port number. Its type is MT SONAME, similar to the mbuf in [Figure 1.6](#). This mbuf is created by the socket layer to return this information to the calling process through the recvfrom or recvmsg system calls. Even though there is room (16 bytes) in the second mbuf on this chain for this socket address structure, it must be stored in its own mbuf since it has a different type (MT SONAME versus MT DATA).

The receiving process is then awakened. If the process is asleep waiting for data to

arrive (which is the scenario in our example), the process is marked as runnable for the kernel to schedule. A process can also be notified of the arrival of data on a socket by the select system call or with the SIGIO signal.

## Process Input

Our process has been asleep in the kernel, blocked in its call to recvfrom, and the process now wakes up. The 26 bytes of data appended to the socket's receive queue by the UDP layer (the received datagram) are copied by the kernel from the mbuf into our program's buffer.

Notice that our program sets the fifth and sixth arguments to recvfrom to null pointers, telling the system call that we're not interested in receiving the sender's IP address and UDP port number. This causes the recvfrom system call to skip the first mbuf in the chain ([Figure 1.11](#)), returning only the 26 bytes of data in the second mbuf. The kernel's recvfrom code then releases the two mbufs in [Figure 1.11](#) and

returns them to its pool of free mbufs.

---

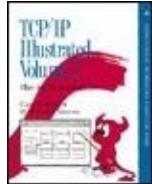
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



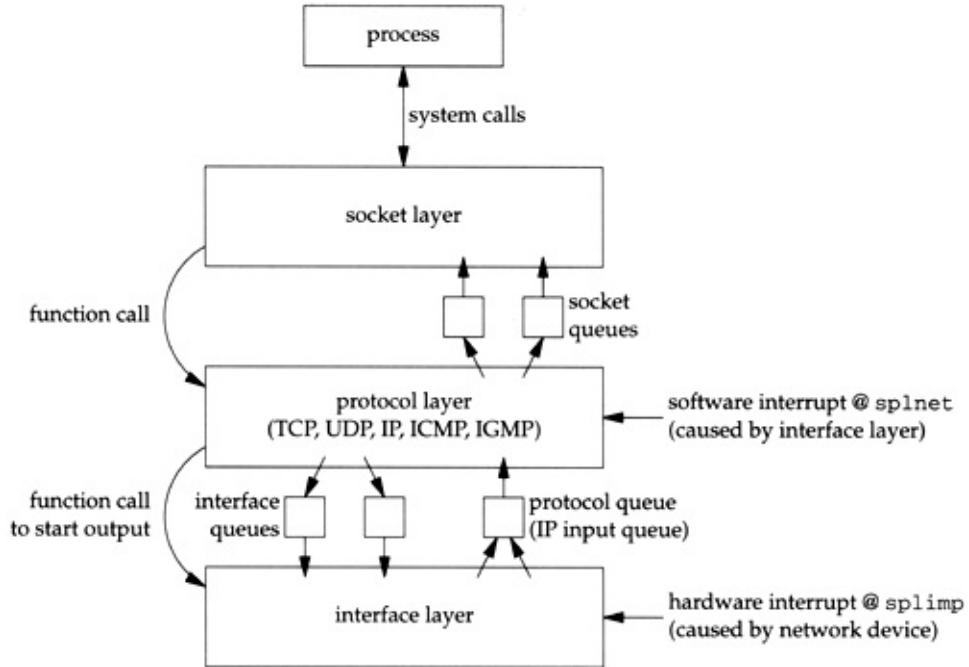
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.11 Network Implementation Overview Revisited

[Figure 1.12](#) summarizes the communication that takes place between the layers for both network output and network input. It repeats [Figure 1.3](#) considering only the Internet protocols and emphasizing the communications between the layers.

**Figure 1.12. Communication between the layers for network input and output.**



The notations `splnet` and `splimp` are discussed in the next section.

We use the plural terms *socket queues* and *interface queues* since there is one queue per socket and one queue per interface (Ethernet, loopback, SLIP, PPP, etc.), but we use the singular term *protocol queue* because there is a single IP input queue. If we considered other protocol layers, we would have one input queue for the XNS protocols and one for the OSI protocols.

## Chapter 1. Introduction

---

### 1.12 Interrupt Levels and Concurrency

We saw in [Section 1.10](#) that the processing of interrupt requests can be either asynchronous and interrupt driven. First, a device driver sends a hardware interrupt to the processor, which posts a software interrupt that is handled by the kernel. When the kernel is finished with these interrupt requests, it returns control to the device driver.

There is a priority level assigned to each hardware interrupt. The normal ordering of the eight priority levels, from highest to lowest, is:

- (all interrupts blocked).

**Figure 1.13. Kernel function priorities**

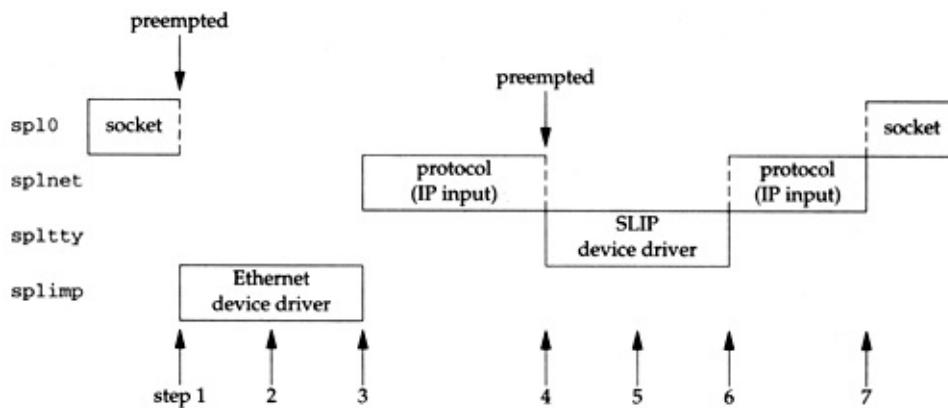
Function	Description
spl0	normal operating mode, nothing blocked
splsoftclock	low-priority clock processing
splnet	network protocol processing
spltty	terminal I/O
splbio	disk and tape I/O
splimp	network device I/O
splclock	high-priority clock processing
splhigh	all interrupts blocked
splx(s)	(see text)

Table 4.5 of [Leffler et al. 1989] shows the priority levels for Net/3 implementation for the 386 uses the eight priority levels. spl0 and splnet are at the same level, and splclock and splhigh are at the same level.

The name *imp* that is used for the network interface message processor (Interface Message Processor), which was the name for the 386.

The ordering of the different priority levels means that a higher-priority interrupt can preempt a lower-priority interrupt. Consider the sequence of events:

**Figure 1.14. Example of priority preemption**



## 1. While the socket layer is executing at step 1, the protocol (IP input) layer begins execution at step 2.

**causing the interface layer to execute a layer code. This is the asynchronous ex**

- While the Ethernet device driver is running, it and schedules a software interrupt to occur at : immediately since the kernel is currently running.
- When the Ethernet device driver completes, the asynchronous execution of the IP input routine.
- A terminal device interrupt occurs (say the co immediately, preempting the protocol layer, sin protocol layer (splnet) in [Figure 1.13](#). This is the routine.
- The SLIP driver places the received packet on software interrupt for the protocol layer.
- When the SLIP driver completes, the preempt processing the packet received from the Ether received from the SLIP driver. Only when there control to whatever it preempted (the socket la
- The socket layer continues from where it was

One concern with these different priority levels different levels. Examples of shared data struct levels in [Figure 1.12](#)the socket, interface, and p routine is taking a received packet off its input

protocol layer, and that device driver can add a data structures (the IP input queue in this example) the interface layer) can be corrupted if nothing

The Net/3 code is sprinkled with calls to the fur paired with a call to splx to return the processo executed by the IP input function at the protocol input queue to process:

```
struct mbuf *m;
int s;

s = splimp();
IF_DEQUEUE(&ipintrq, m);
splx(s);

if (m == 0)
    return;
```

The call to splimp raises the CPU priority to the any network device driver interrupt from occurring value of the function and stored in the variable remove the next packet at the head of the IP input chain in the variable m. Finally the CPU priority called, by calling splx with an argument of s (th

Since all network device driver interrupts are d

amount of code between these calls should be |  
period of time, additional device interrupts coul  
the test of the variable m (to see if there is anc  
splx, and not before the call.

The Ethernet output routine needs these spl ca  
interface's queue, tests whether the interface is  
busy.

```
struct mbuf *m;
int s;

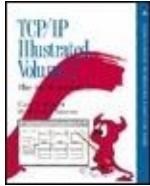
s = splimp();
/*
 * Queue message on interface, e
 */
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);      /* 
    splx(s);
    error = ENOBUFS;
    goto bad;
}

IF_ENQUEUE(&ifp->if_snd, m); /* 
if ((ifp->if_flags & IFF_OACTIVE
    (*ifp->if_start)(ifp); /*
```

```
splx(s);
```

The reason device interrupts are disabled in this function is because the next packet off its send queue while the processor is waiting for interrupt acknowledgement. The driver's send queue is a data structure shared by all interrupt handlers.

We'll see calls to the spl functions throughout the code.



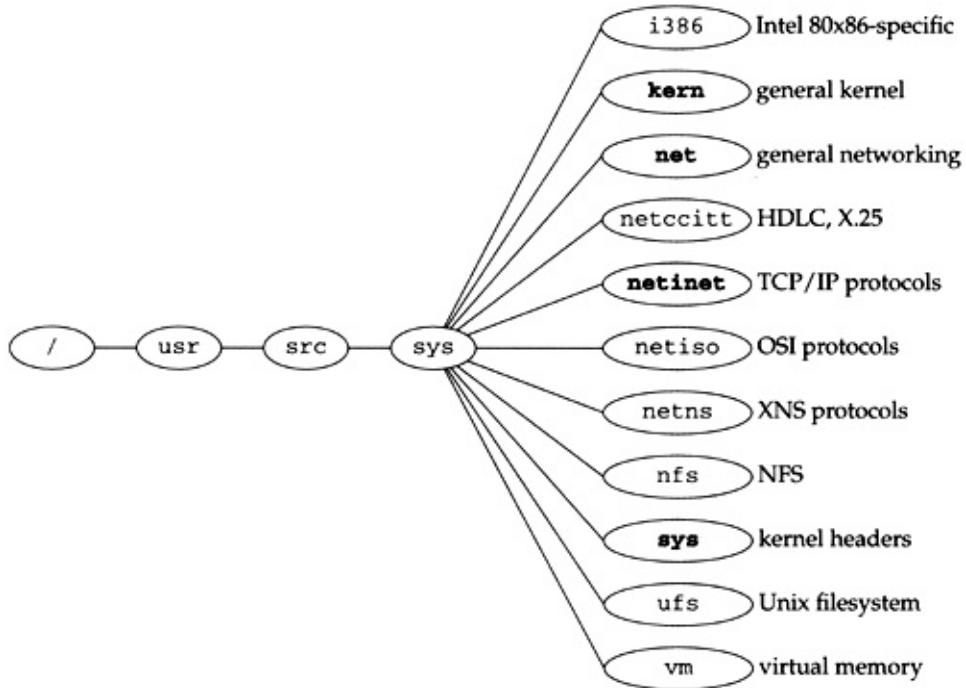
[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.13 Source Code Organization

Figure 1.15 shows the organization of the Net/3 networking source tree, assuming it is located in the /usr/src/sys directory.

**Figure 1.15. Net/3 source code organization.**



This text focuses on the `netinet` directory, which contains all the TCP/IP source code. We also look at some files in the `kern` and `net` directories. The former contains the protocol-independent socket code, and the latter contains some general networking functions used by the TCP/IP routines, such as the routing code.

Briefly, the files contained in each directory are as follows:

- `i386`: the Intel 80x86-specific directories. For example, the directory `i386/isa` contains the device drivers specific to the ISA bus. The directory

i386/stand contains the stand-alone bootstrap code.

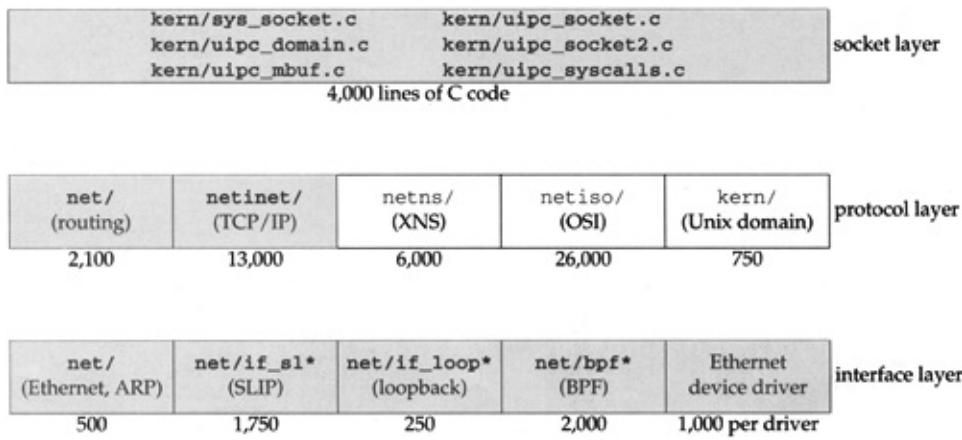
- kern: general kernel files that don't belong in one of the other directories. For example, the kernel files to handle the fork and exec system calls are in this directory. We look at only a few files in this directorythe ones for the socket system calls (the socket layer in [Figure 1.3](#)).
- net: general networking files, for example, general network interface functions, the BPF (BSD Packet Filter) code, the SLIP driver, the loopback driver, and the routing code. We look at some of the files in this directory.
- netccitt: interface code for the OSI protocols, including the HDLC (high-level data-link control) and X.25 drivers.
- netinet: the code for the Internet protocols: IP, ICMP, IGMP, TCP, and UDP. This text focuses on the files in this directory.

- netiso: the OSI protocols.
- netns: the Xerox XNS protocols.
- nfs: code for Sun's Network File System.
- sys: system headers. We look at several headers in this directory. The files in this directory also appear in the directory /usr/include/sys.
- ufs: code for the Unix filesystem, sometimes called the *Berkeley fast filesystem*. This is the normal disk-based filesystem.
- vm: code for the virtual memory system.

[Figure 1.16](#) gives another view of the source code organization, this time mapped to our three kernel layers. We ignore directories such as netimp and nfs that we don't consider in this text.

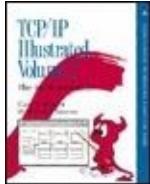
**Figure 1.16. Net/3 source code organization mapped to three kernel**

# layers.



The numbers below each box are the approximate number of lines of C code for that feature, which includes all comments in the source files.

We don't look at all the source code shown in this figure. The netns and netiso directories are shown for comparison against the Internet protocols. We only consider the shaded boxes.



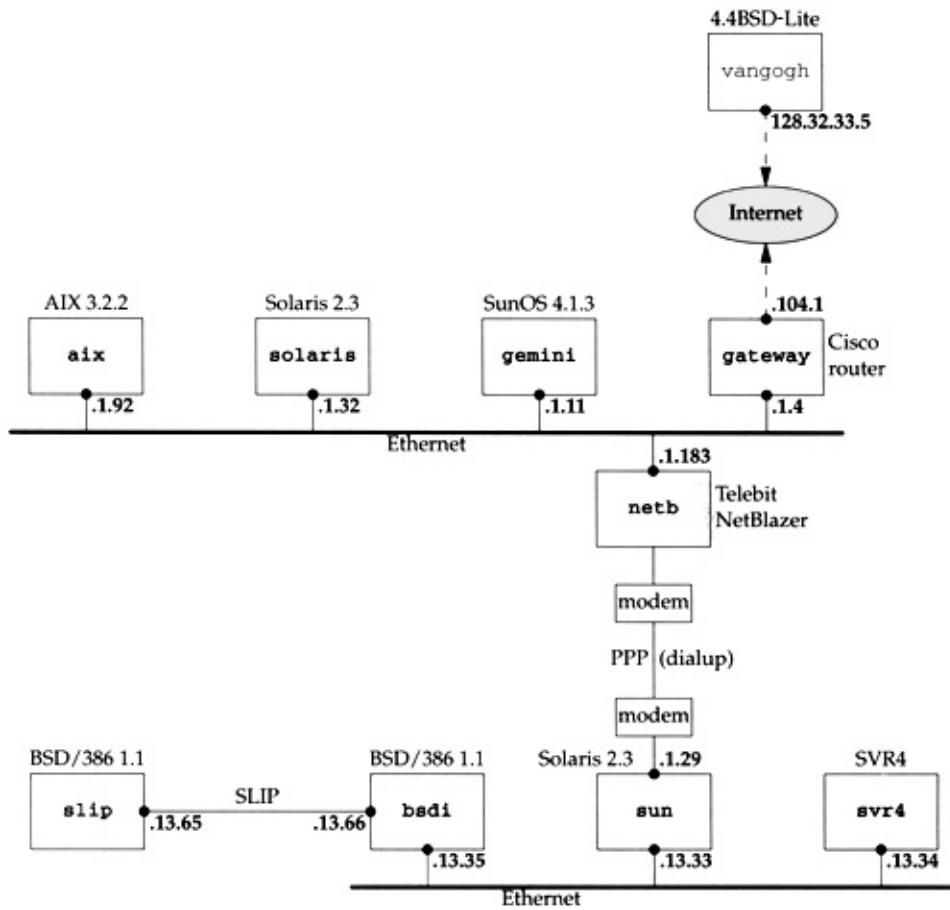
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.14 Test Network

Figure 1.17 shows the test network that is used for all the examples in the text. Other than the host vangogh at the top of the figure, all the IP addresses belong to the class B network ID 140.252, and all the hostnames belong to the .tuc.noao.edu domain. (noao stands for "National Optical Astronomy Observatories" and tuc stands for Tucson.) For example, the system in the lower right has a complete hostname of svr4.tuc.noao.edu and an IP address of 140.252.13.34. The notation at the top of each box is the operating system running on that system.

**Figure 1.17. Test network used for all the examples in the text.**



The host at the top has a complete name of vangogh.cs.berkeley.edu and is reachable from the other hosts across the Internet.

This figure is nearly identical to the test network used in Volume 1, although some of the operating systems have been

upgraded and the dialup link between sun and netb now uses PPP instead of SLIP. Additionally, we have replaced the Net/2 networking code provided with BSD/386 V1.1 with the Net/3 networking code.

---

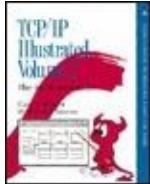
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 1. Introduction

### 1.15 Summary

This chapter provided an overview of the Net/3 networking code. Using a simple program ([Figure 1.2](#)) that sends a UDP datagram to a daytime server and receives a reply, we've followed the resulting output and input through the kernel. Mbufs hold the information being output and the received IP datagrams. The next chapter examines mbufs in more detail.

UDP output occurs when the process executes the `sendto` system call, while IP input is asynchronous. When an IP datagram is received by a device driver, the datagram is placed onto IP's input queue and a software interrupt is

scheduled to cause the IP input function to execute. We reviewed the different interrupt levels used by the networking code within the kernel. Since many of the networking data structures are shared by different layers that can execute at different interrupt priorities, the code must be careful when accessing or modifying these shared structures. We'll encounter calls to the spl functions in almost every function that we look at.

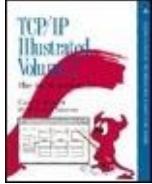
The chapter finishes with a look at the overall organization of the source code in Net/3, focusing on the code that this text examines.

## Exercises

Type in the example program ([Figure 1.2](#)) and run it on your system. If your system has a system call tracing **1.1** capability, such as trace (SunOS 4.x), truss (SVR4), or ktrace (4.4BSD), use it to determine the system calls invoked by this example.

In our example that calls IF\_DEQUEUE in [Section 1.12](#), we noted that the call to splimp blocks network device **1.2** drivers from interrupting. While Ethernet drivers execute at this level, what happens to SLIP drivers?

---



[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Chapter 2. Mbufs: Memory Buffers

Section 2.1. Introduction

Section 2.2. Code Introduction

Section 2.3. Mbuf Definitions

Section 2.4. mbuf Structure

Section 2.5. Simple Mbuf Macros and Functions

Section 2.6. m\_devget and m\_pullup Functions

Section 2.7. Summary of Mbuf Macros and Functions

Section 2.8. Summary of Net/3 Networking Data Structures

Section 2.9. m\_copy and Cluster Reference Counts

Section 2.10. Alternatives

## Section 2.11. Summary

---

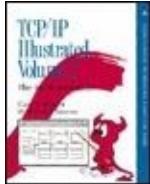
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.1 Introduction

Networking protocols place many demands on the memory management facilities of the kernel. These demands include easily manipulating buffers of varying sizes, prepending and appending data to the buffers as the lower layers encapsulate data from higher layers, removing data from buffers (as headers are removed as data packets are passed up the protocol stack), and minimizing the amount of data copied for all these operations. The performance of the networking protocols is directly related to the memory management scheme used within the kernel.

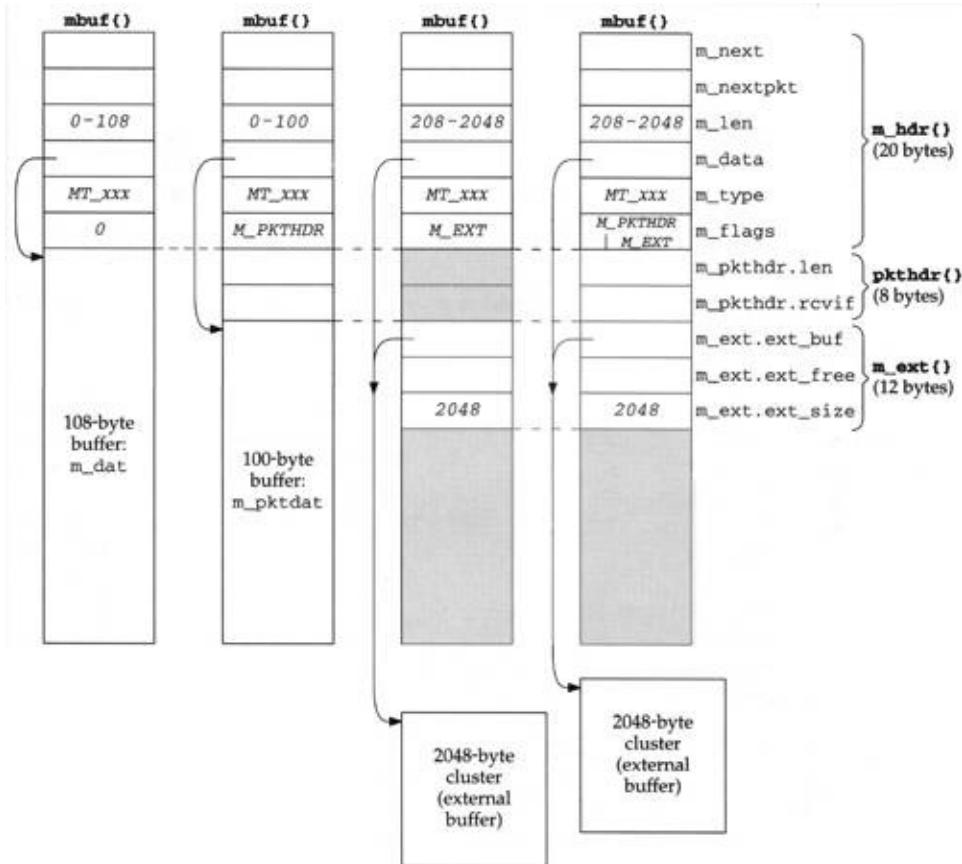
In [Chapter 1](#) we introduced the memory buffer used throughout the Net/3 kernel: the *mbuf*, which is an abbreviation for "memory buffer." In this chapter we look in more detail at mbufs and at the functions within the kernel that are used to manipulate them, as we will encounter mbufs on almost every page of the text. Understanding mbufs is essential for understanding the rest of the text.

The main use of mbufs is to hold the user data that travels from the process to the network interface, and vice versa. But mbufs are also used to contain a variety of other miscellaneous data: source and destination addresses, socket options, and so on.

[Figure 2.1](#) shows the four different kinds of mbufs that we'll encounter, depending on the M\_PKTHDR and M\_EXT flags in the *m\_flags* member. The differences between the four mbufs in [Figure 2.1](#), from left to right, are as follows:

**Figure 2.1. Four different types of mbufs,**

depending on the `m_flags` value.



1. If `m_flags` equals 0, the `mbuf` contains only data. There is room in the `mbuf` for up to 108 bytes of data (the `m_dat` array). The `m_data` pointer points somewhere in this 108-byte buffer. We show it pointing to the start of the buffer, but it can point anywhere in the buffer. The `m_len` member specifies the number of bytes of data,

**starting at m\_data. Figure 1.6 was an example of this type of mbuf.**

**In Figure 2.1 there are six members in the m\_hdr structure, and its total size is 20 bytes. When we look at the C definition of this structure (Figure 2.8) we'll see that the first four members occupy 4 bytes each and the last two occupy 2 bytes each. We don't try to differentiate between the 4-byte members and the 2-byte members in Figure 2.1.**

- The second type of mbuf has an m\_flags value of M\_PKTHDR, specifying a *packet header*, that is, the first mbuf describing a packet of data. The data is still contained within the mbuf itself, but because of the 8 bytes taken by the packet header, only 100 bytes of data fit within this mbuf (in the m\_pktdat array). Figure 1.10 was an example of this type of mbuf.

The m\_pkthdr.len value is the total length of all the data in the mbuf chain for this packet: the sum of the m\_len values for all the mbufs linked through the m\_next

pointer, as shown in [Figure 1.8](#). The `m_pkthdr.rcvif` member is not used for output packets, but for received packets it contains a pointer to the received interface's `ifnet` structure ([Figure 3.6](#)).

- The next type of mbuf does not contain a packet header (`M_PKTHDR` is not set) but contains more than 208 bytes of data, so an external buffer called a *cluster* is used (`M_EXT` is set). Room is still allocated in the mbuf itself for the packet header structure, but it is unused we show it shaded in [Figure 2.1](#). Instead of using multiple mbufs to contain the data (the first with 100 bytes of data, and all the rest with 108 bytes of data each), Net/3 allocates a cluster of size 1024 or 2048 bytes. The `m_data` pointer in the mbuf points somewhere inside this cluster.

The Net/3 release supports seven different architectures. Four define the size of a cluster as 1024 bytes (the traditional value) and three define it as 2048. The reason 1024 has been used historically is to save memory: if the cluster size is 2048, about one-quarter of each cluster is

unused for Ethernet packets (1500 bytes maximum). We'll see in [Section 27.5](#) that the Net/3 TCP never sends more than the cluster size per TCP segment, so with a cluster size of 1024, almost one-third of each 1500-byte Ethernet frame is unused. But [[Mogul 1993, Figure 15.15](#)] shows that a sizable performance improvement occurs on an Ethernet when maximum-sized frames are sent instead of 1024-byte frames. This is a performance-versus-memory tradeoff. Older systems used 1024-byte clusters to save memory while newer systems with cheaper memory use 2048 to increase performance. Throughout this text we assume a cluster size of 2048.

Unfortunately different names have been used for what we call *clusters*. The constant MCLBYTES is the size of these buffers (1024 or 2048) and the names of the macros to manipulate these buffers are MCLGET, MCLALLOC, and MCLFREE. This is why we call them *clusters*. But we also see that the mbuf flag is M\_EXT, which stands for "external" buffer. Finally, [[Leffler et al. 1989](#)] calls them *mapped pages*. This

latter name refers to their implementation, and we'll see in [Section 2.9](#) that clusters can be shared when a copy is required.

We would expect the minimum value of `m_len` to be 209 for this type of mbuf, not 208 as we indicate in the figure. That is, a record with 208 bytes of data can be stored in two mbufs, with 100 bytes in the first and 108 in the second. The source code, however, has a bug and allocates a cluster if the size is greater than or equal to 208.

- The final type of mbuf contains a packet header and contains more than 208 bytes of data. Both `M_PKTHDR` and `M_EXT` are set.

There are numerous additional points we need to make about [Figure 2.1](#):

- The size of the mbuf structure is always 128 bytes. This means the amount of unused space following the `m_ext` structure in the two mbufs on the right in [Figure 2.1](#) is 88 bytes (128 20 8

12).

- A data buffer with an `m_len` of 0 bytes is OK since some protocols (e.g., UDP) allow 0-length records.
- In each of the mbufs we show the `m_data` member pointing to the beginning of the corresponding buffer (either the mbuf buffer itself or a cluster). This pointer can point anywhere in the corresponding buffer, not necessarily the front.
- Mbufs with a cluster always contain the starting address of the buffer (`m_ext.ext_buf`) and its size (`m_ext.ext_size`). We assume a size of 2048 throughout this text. The `m_data` and `m_ext. ext_buf` members are not the same (as we show) unless `m_data` also points to the first byte of the buffer. The third member of the `m_ext` structure, `ext_free`, is not currently used by Net/3.
- The `m_next` pointer links together the mbufs forming a single packet (record)

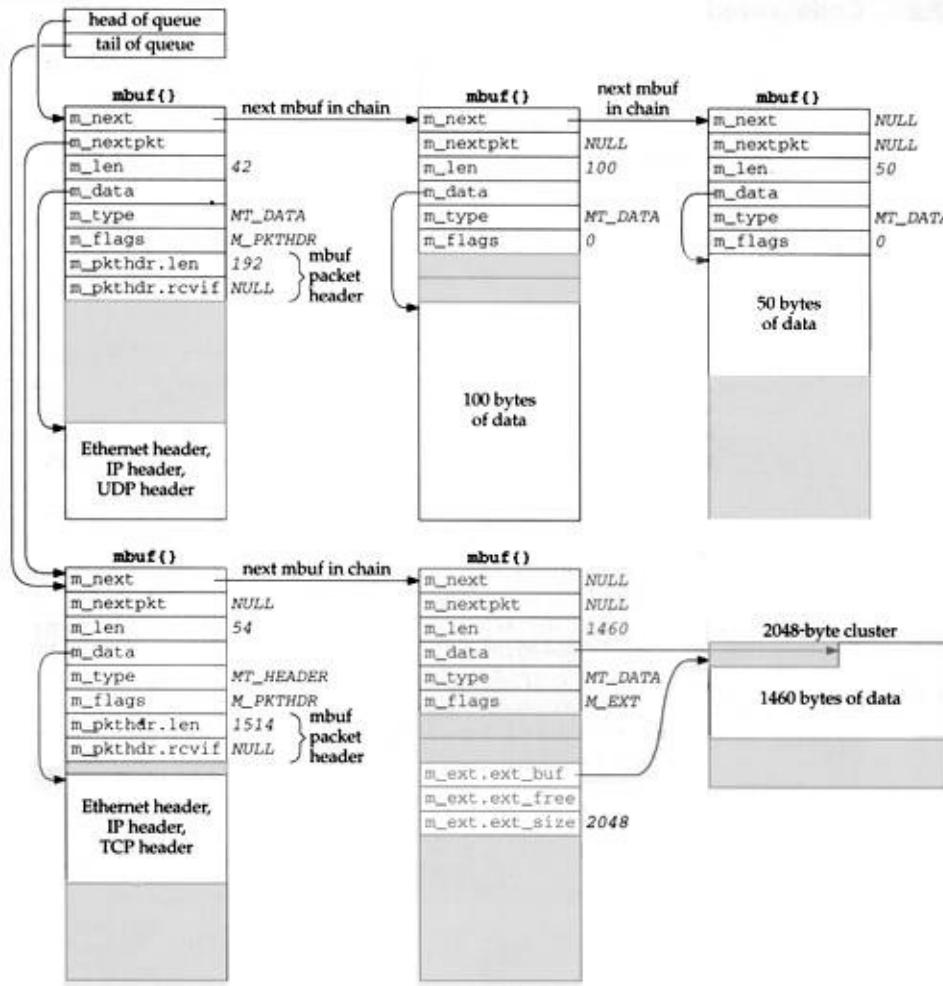
into an *mbuf chain*, as in [Figure 1.8](#).

- The `m_nextpkt` pointer links multiple packets (records) together to form a *queue of mbufs*. Each packet on the queue can be a single mbuf or an mbuf chain. The first mbuf of each packet contains a packet header. If multiple mbufs define a packet, the `m_nextpkt` member of the first mbuf is the only one used; the `m_nextpkt` member of the remaining mbufs on the chain are all null pointers.

[Figure 2.2](#) shows an example of two packets on a queue. It is a modification of [Figure 1.8](#). We have placed the UDP datagram onto the interface output queue (showing that the 14-byte Ethernet header has been prepended to the IP header in the first mbuf on the chain) and have added a second packet to the queue: a TCP segment containing 1460 bytes of user data. The TCP data is contained in a cluster and an mbuf has been prepended to contain its Ethernet, IP, and TCP headers. With the cluster we show that the data pointer into the cluster (`m_data`)

need not point to the front of the cluster. We show that the queue has a head pointer and a tail pointer. This is how the interface output queues are handled in Net/3. We have also added the m\_ext structure to the mbuf with the M\_EXT flag set and have shaded in the unused pkthdr structure of this mbuf.

**Figure 2.2. Two packets on a queue: first with 192 bytes of data and second with 1514 bytes of data.**

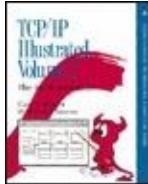


The first mbuf with the packet header for the UDP datagram has a type of MT\_DATA, but the first mbuf with the packet header for the TCP segment has a type of MT\_HEADER. This is a side effect of the different way UDP and TCP prepend the headers to their data, and makes no difference. Mbufs of these two types are essentially the same. It is the m\_flags value of M\_PKTHDR in the first mbuf on the chain that indicates a

packet header.

Careful readers may note a difference between our picture of an mbuf (the Net/3 mbuf, [Figure 2.1](#)) and the picture in [[Leffler et al. 1989](#), p. 290], a Net/1 mbuf. The changes were made in Net/2: adding the `m_flags` member, renaming the `m_act` pointer to be `m_nextpkt`, and moving this pointer to the front of the mbuf.

The difference in the placement of the protocol headers in the first mbuf for the UDP and TCP examples is caused by UDP calling `M_PREPEND` ([Figure 23.15](#) and [Exercise 23.1](#)) while TCP calls `MGETHDR` ([Figure 26.25](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

## 2.2 Code Introduction

The mbuf functions are in a single C file and the mbuf macros and various mbuf definitions are in a single header, as shown in [Figure 2.3](#).

**Figure 2.3. Files discussed in this chapter.**

File	Description
<code>sys/mbuf.h</code>	mbuf structure, mbuf macros and definitions
<code>kern/uipc_mbuf.c</code>	mbuf functions

## Global Variables

One global variable is introduced in this

chapter, shown in [Figure 2.4](#).

## Figure 2.4. Global variables introduced in this chapter.

Variable	Datatype	Description
mbstat	struct mbstat	mbuf statistics ( <a href="#">Figure 2.5</a> )

## Statistics

Various statistics are maintained in the global structure mbstat, described in [Figure 2.5](#).

## Figure 2.5. Mbuf statistics maintained in the mbstat structure.

mbstat member	Description
m_clfree	#free clusters
m_clusters	#clusters obtained from page pool
m_drain	#times protocol's drain functions called to reclaim space
m_drops	#times failed to find space (not used)
m_mbufs	#mbufs obtained from page pool (not used)
m_mtotypes [256]	counter of current mbuf allocations: MT_xxx index
m_spare	spare field (not used)
m_wait	#times waited for space (not used)

This structure can be examined with the

netstat -m command; [Figure 2.6](#) shows some sample output. The two values printed for the number of mapped pages in use are m\_clusters (34) minus m\_clfree (32), giving the number of clusters currently in use (2), and m\_clusters (34).

**Figure 2.6. Sample mbuf statistics.**

netstat -m output	mbstat member
99 mbufs in use:	
1 mbufs allocated to data	m_mtotypes[MT_DATA]
43 mbufs allocated to packet headers	m_mtotypes[MT_HEADER]
17 mbufs allocated to protocol control blocks	m_mtotypes[MT_PCB]
20 mbufs allocated to socket names and addresses	m_mtotypes[MT SONAME]
18 mbufs allocated to socket options	m_mtotypes[MT_SOOPTS]
2/34 mapped pages in use	(see text)
80 Kbytes allocated to network (20% in use)	(see text)
0 requests for memory denied	m_drops
0 requests for memory delayed	m_wait
0 calls to protocol drain routines	m_drain

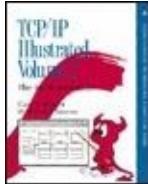
The number of Kbytes of memory allocated to the network is the mbuf memory ( $99 \times 128$  bytes) plus the cluster memory ( $34 \times 2048$  bytes) divided by 1024. The percentage in use is the mbuf memory ( $99 \times 128$  bytes) plus the cluster memory in use ( $2 \times 2048$  bytes) divided by the total network memory (80 Kbytes), times 100.

## Kernel Statistics

The mbuf statistics show a common technique that we see throughout the Net/3 sources. The kernel keeps track of certain statistics in a global variable (the mbstat structure in this example). A process (in this case the netstat program) examines the statistics while the kernel is running.

Rather than provide system calls to fetch the statistics maintained by the kernel, the process obtains the address within the kernel of the data structure in which it is interested by reading the information saved by the link editor when the kernel was built. The process then calls the kvm(3) functions to read the corresponding location in the kernel's memory by using the special file /dev/mem. If the kernel's data structure changes from one release to the next, any program that reads that structure must also change.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.3 Mbuf Definitions

There are a few constants that we encounter repeatedly when dealing with mbufs. Their values are shown in [Figure 2.7](#). All are defined in `mbuf.h` except `MCLBYTES`, which is defined in `/usr/include/machine/param.h`.

**Figure 2.7. Mbuf constants from `mbuf.h`.**

Constant	Value (#bytes)	Description
<code>MCLBYTES</code>	2048	size of an mbuf cluster (external buffer)
<code>MHLEN</code>	100	max amount of data in mbuf with packet header
<code>MINCLSIZE</code>	208	smallest amount of data to put into cluster
<code>MLEN</code>	108	max amount of data in normal mbuf
<code>MSIZE</code>	128	size of each mbuf

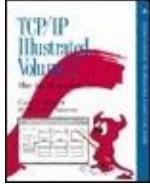
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.4 mbuf Structure

Figure 2.8 shows the definition of the mbuf structure.

**Figure 2.8. Mbuf structures.**

graphics/02fig08.jpg

The mbuf structure is defined as an m\_hdr structure, followed by a union. As the comments indicate, the contents of the union depend on the flags M\_PKTHDR and M\_EXT.

93-103

These 11 #define statements simplify access to the members of the structures and unions within the mbuf structure. We will see this technique used throughout the Net/3 sources whenever we encounter a structure containing other structures or unions.

We previously described the purpose of the first two members in the mbuf structure: the m\_next pointer links mbufs together into an mbuf chain and the m\_nextpkt pointer links mbuf chains together into a *queue of mbufs*.

[Figure 1.8](#) differentiated between the m\_len member of each mbuf and the m\_pkthdr.len member in the packet header. The latter is the sum of all the m\_len members of all the mbufs on the chain.

There are five independent values for the m\_flags member, shown in [Figure 2.9](#).

**Figure 2.9. m\_flags values.**

<code>m_flags</code>	Description
<code>M_BCAST</code>	sent/received as link-level broadcast
<code>M_EOR</code>	end of record
<code>M_EXT</code>	cluster (external buffer) associated with this mbuf
<code>M_MCAST</code>	sent/received as link-level multicast
<code>M_PKTHDR</code>	first mbuf that forms a packet (record)
<code>M_COPYFLAGS</code>	<code>M_PKTHDR</code> / <code>M_EOR</code> / <code>M_BCAST</code> / <code>M_MCAST</code>

We have already described the `M_EXT` and `M_PKTHDR` flags. `M_EOR` is set in an mbuf containing the end of a record. The Internet protocols (e.g., TCP) never set this flag, since TCP provides a byte-stream service without any record boundaries. The OSI and XNS transport layers, however, do use this flag. We will encounter this flag in the socket layer, since this layer is protocol independent and handles data to and from all the transport layers.

The next two flags, `M_BCAST` and `M_MCAST`, are set in an mbuf when the packet will be sent to or was received from a link-layer broadcast address or multicast address. These two constants are flags between the protocol layer and the interface layer ([Figure 1.3](#)).

The final value, `M_COPYFLAGS`, specifies the flags that are copied when an mbuf

containing a packet header is copied.

[Figure 2.10](#) shows the MT\_XXX constants used in the m\_type member to identify the type of data stored in the mbuf. Although we tend to think of an mbuf as containing user data that is sent or received, mbufs can contain a variety of different data structures. Recall in [Figure 1.6](#) that an mbuf was used to hold a socket address structure with the destination address for the sendto system call. Its m\_type member was set to MT SONAME.

## Figure 2.10. Values for m\_type member.

Mbuf m_type	Used in Net/3 TCP/IP code	Description	Memory type
MT_CONTROL	•	extra-data protocol message	M_MBUF
MT_DATA	•	dynamic data allocation	M_MBUF
MT_FREE		should be on free list	M_FREE
MT_FTABLE	•	fragment reassembly header	M_FTABLE
MT_HEADER	•	packet header	M_MBUF
MT_HTABLE		IMP host tables	M_HTABLE
MT_IFADDR		interface address	M_IFADDR
MT_OOBDATA		expedited (out-of-band) data	M_MBUF
MT_PCB		protocol control block	M_PCB
MT_RIGHTS		access rights	M_MBUF
MT_RTABLE		routing tables	M_RTABLE
MT SONAME	•	socket name	M_MBUF
MT_SOOPTS	•	socket options	M_SOOPTS
MT_SOCKET		socket structure	M_SOCKET

Not all of the mbuf type values in [Figure 2.10](#) are used in Net/3. Some are historical (MT\_HTABLE), and others are not used in the TCP/IP code but are used elsewhere in the kernel. For example, MT\_OOBDATA is used by the OSI and XNS protocols, but TCP handles out-of-band data differently (as we describe in [Section 29.7](#)). We describe the use of other mbuf types when we encounter them later in the text.

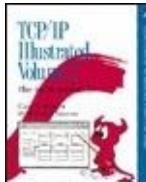
The final column of this figure shows the M\_xxx values associated with the piece of memory allocated by the kernel for the different types of mbufs. There are about 60 possible M\_xxx values assigned to the different types of memory allocated by the kernel's malloc function and MALLOC macro. [Figure 2.6](#) showed the mbuf allocation statistics from the netstat -m command including the counters for each MT\_xxx type. The vmstat -m command shows the kernel's memory allocation statistics including the counters for each M\_xxx type.

Since mbufs have a fixed size (128

bytes) there is a limit for what an mbuf can be used for the data contents cannot exceed 108 bytes. Net/2 used an mbuf to hold a TCP protocol control block (which we cover in [Chapter 24](#)), using the mbuf type of MT\_PCB. But 4.4BSD increased the size of this structure from 108 bytes to 140 bytes, forcing the use of a different type of kernel memory allocation for the structure.

Observant readers may have noticed that in [Figure 2.10](#) we say that mbufs of type MT\_PCB are not used, yet [Figure 2.6](#) shows a nonzero counter for this type. The Unix domain protocols use this type of mbuf, and it is important to remember that the statistics are for mbuf usage across all protocol suites, not just the Internet protocols.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.5 Simple Mbuf Macros and Functions

There are more than two dozen macros and functions that deal with mbufs (allocate an mbuf, free an mbuf, etc.). We look at the source code for only a few of the macros and functions, to show how they're implemented.

Some operations are provided as both a macro and function. The macro version has an uppercase name that begins with M, and the function has a lowercase name that begins with m\_. The difference in the two is the standard time-versus-space tradeoff. The macro version is expanded inline by the C preprocessor each time it is used (requiring more code space), but it executes faster since it doesn't require a

function call (which can be expensive on some architectures). The function version, on the other hand, becomes a few instructions each time it is invoked (push the arguments onto the stack, call the function, etc.), taking less code space but more execution time.

## m\_get Function

We'll look first at the function that allocates an mbuf: m\_get, shown in [Figure 2.11](#). This function merely expands the macro MGET.

**Figure 2.11. m\_get function: allocate an mbuf**

```
134 struct mbuf *
135 m_get(nowait, type)
136 int     nowait, type;
137 {
138     struct mbuf *m;
139     MGET(m, nowait, type);
140     return (m);
141 }
```

Notice that the Net/3 code does not use ANSI C argument declarations. All the Net/3 system headers, however, *do* provide ANSI C function prototypes for all kernel functions, if an ANSI C compiler is being used. For example, the <sys/mbuf.h> header includes

the line

```
struct mbuf *m_get (int, int);
```

These function prototypes provide compile-time checking of the arguments and return values whenever a kernel function is called.

The caller specifies the nowait argument as either M\_WAIT or M\_DONTWAIT, depending whether it wants to wait if the memory is not available. As an example of the difference, when the socket layer asks for an mbuf to store the destination address of the sendto system call ([Figure 1.6](#)) it specifies M\_WAIT, since blocking at this point is OK. But when the Ethernet device driver asks for an mbuf to store a received frame ([Figure 1.10](#)) it specifies M\_DONTWAIT, since it is executing as a device interrupt handler and cannot be put to sleep waiting for an mbuf. In this case it is better for the device driver to discard the Ethernet frame if the memory is not available.

## MGET Macro

[Figure 2.12](#) shows the MGET macro. A call to

MGET to allocate the mbuf to hold the destination address for the sendto system call ([Figure 1.6](#)) might look like

## Figure 2.12. MGET macro.

```
154 #define MGET(m, how, type) { \
155     MALLOC((m), struct mbuf *, MSIZE, mbtypes[type], (how)); \
156     if ((m)) { \
157         (m)->m_type = (type); \
158         MBUFLLOCK(mbstat.m_mtypes[type]++); \
159         (m)->m_next = (struct mbuf *)NULL; \
160         (m)->m_nextpkt = (struct mbuf *)NULL; \
161         (m)->m_data = (m)->m_dat; \
162         (m)->m_flags = 0; \
163     } else \
164         (m) = m_retry((how), (type)); \
165 }
```

```
MGET (m, M_WAIT, MT SONAME);  
if (m == NULL)  
    return (ENOBUFS);
```

Even though the caller specifies M\_WAIT, the return value must still be checked, since, as we see in [Figure 2.13](#), waiting for an mbuf does not guarantee that one will be available.

## Figure 2.13. m\_retry function.

```
92 struct mbuf *
93 m_retry(i, t)
94 int i, t;
95 {
96     struct mbuf *m;
97     m_reclaim();
98 #define m_retry(i, t)  (struct mbuf *)0
99     MGET(m, i, t);
100 #undef m_retry
101     return (m);
102 }
```

uipc\_mbuf.c

## 154-157

MGET first calls the kernel's MALLOC macro, which is the general-purpose kernel memory allocator. The array mbtypes converts the mbuf MT\_xxx value into the corresponding M\_xxx value ([Figure 2.10](#)). If the memory can be allocated, the m\_type member is set to the argument's value.

## 158

The kernel structure that keeps mbuf statistics for each type of mbuf is incremented (mbstat). The macro MBUFLOCK changes the processor priority ([Figure 1.13](#)) while executing the statement specified as its argument, and then resets the priority to its previous value. This prevents network device interrupts from occurring while the statement mbstat.m\_mttype [type]++; is executing, because mbufs can be allocated at various layers within the kernel.

Consider a system that implements the `++` operator in C using three steps: (1) load the current value into a register, (2) increment the register, and (3) store the register into memory. Assume the counter's value is 77 and MGET is executing at the socket layer. Assume steps 1 and 2 are executed (the register's value is 78) and a device interrupt occurs. If the device driver also executes MGET for the same type of mbuf, the value in memory is fetched (77), incremented (78), and stored back into memory. When step 3 of the interrupted execution of MGET resumes, it stores its register (78) into memory. But the counter should be 79, not 78, so the counter has been corrupted.

## 159-160

The two mbuf pointers, `m_next` and `m_nextpkt` are set to null pointers. It is the caller's responsibility to add the mbuf to a chain or queue, if necessary.

## 161-162

Finally the data pointer is set to point to the beginning of the 108-byte mbuf buffer and the flags are set to 0.

163-164

If the call to the kernel's memory allocator fails `m_retry` is called ([Figure 2.13](#)). The first argument is either `M_WAIT` or `M_DONTWAIT`.

## `m_retry` Function

[Figure 2.13](#) shows the `m_retry` function.

92-97

The first function called by `m_retry` is `m_reclaim`. We'll see in [Section 7.4](#) that each protocol can define a "drain" function to be called by `m_reclaim` when the system gets low on available memory. We'll also see in [Figure 10.32](#) that when IP's drain function is called, all IP fragments waiting to be reassembled into IP datagrams are discarded. TCP's drain function does nothing and UDP doesn't even define a drain function.

98-102

Since there's a chance that more memory *might* be available after the call to `m_reclaim`, the `MGET` macro is called again, to try to obtain the

mbuf. Before expanding the MGET macro (Figure 2.12), m\_retry is defined to be a null pointer. This prevents an infinite loop if the memory still isn't available: the expansion of MGET will set r to this null pointer instead of calling the m\_retry function. After the expansion of MGET, this temporary definition of m\_retry is undefined, in case there is another reference to MGET later in the source file.

## Mbuf Locking

In the functions and macros that we've looked in this section, other than the call to MBUFLOC in Figure 2.12, there are no calls to the spl functions to protect these functions and macros from being interrupted. What we haven't shown however, is that the macro MALLOC contains an splimp at the beginning and an splx at the end. The macro MFREE contains the same protection. Mbufs are allocated and released at all layers within the kernel, so the kernel must protect the data structures that it uses for memory allocation.

Additionally, the macros MCLALLOC and MCLFREE, which allocate and release an mbuf

cluster, are surrounded by an splimp and an splx, since they modify a linked list of available clusters.

Since the memory allocation and release macrc along with the cluster allocation and release macros are protected from interrupts, we normally do not encounter calls to the spl functions around macros and functions such as MGET and m\_get.

## Chapter 2. Mbufs: Memory Buffers

---

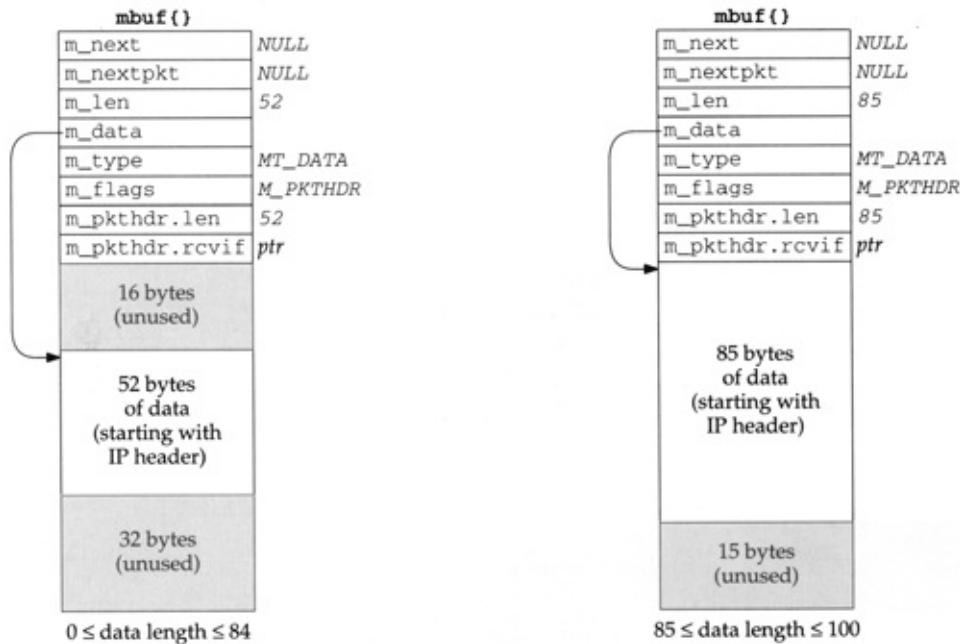
### 2.6 m\_devget and m\_pullup Functions

We encounter the `m_pullup` function when we receive frames for UDP, and TCP. It is called to guarantee that the bytes following the end of the corresponding protocol header (e.g., IP header) are contiguous. If `m_pullup` is called with a value of zero, otherwise the specified number of bytes are copied from the start of the frame to a buffer that is guaranteed to be contiguous. To understand the usage of `m_pullup`, it is necessary to understand its implementation and its interaction with both the `m_copyin` and `dtom` macros. This description also provides some insight into the use of mbufs in Net/3.

#### `m_devget` Function

When an Ethernet frame is received, the device driver calls the `m_devget` function to create an mbuf chain and copy the frame from memory. Depending on the length of the received frame, there are four different possibilities for the resulting mbuf chain. These possibilities are shown in [Figure 2.14](#).

**Figure 2.14. First two types of mbuf**



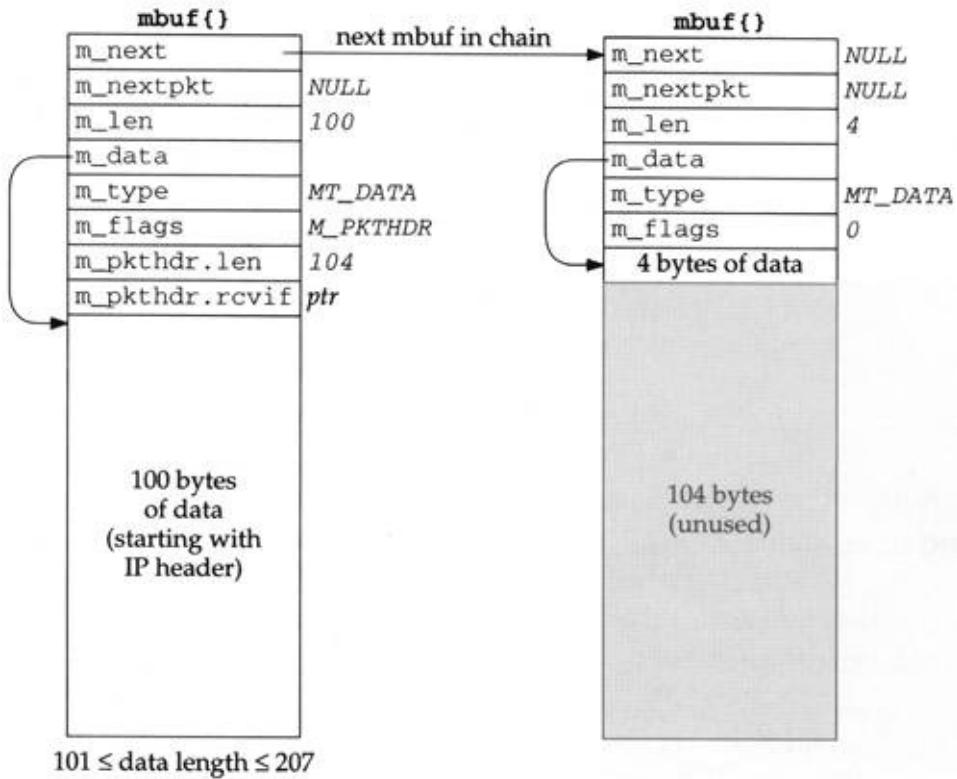
1. The left mbuf in Figure 2.14 is used when the data length is between 0 and 84 bytes. In this figure we see the first two types of mbufs. The first type contains data: a 20-byte IP header and a 32-byte TCP header plus 12 bytes of TCP options. The second type contains the data in the mbuf returned by m\_devget. Note that the realistic minimum value for m\_len is 8 bytes for a UDP header, and a 0-length header is not possible.

m\_devget leaves 16 bytes unused at the end of the data. Although the 14-byte Ethernet header is not allocated for a 14-byte Ethernet header, it can be used for output. We'll encounter this response by using the received mbuf as input to m\_copyin.

**icmp\_reflect and tcp\_respond. In both cases, the amount of data in a datagram is normally less than 84 bytes. There is therefore room for 16 bytes at the front, which can be used for the outgoing datagram. The reason 16 bytes are needed is that the IP header longword must have the IP header longword aligned in memory.**

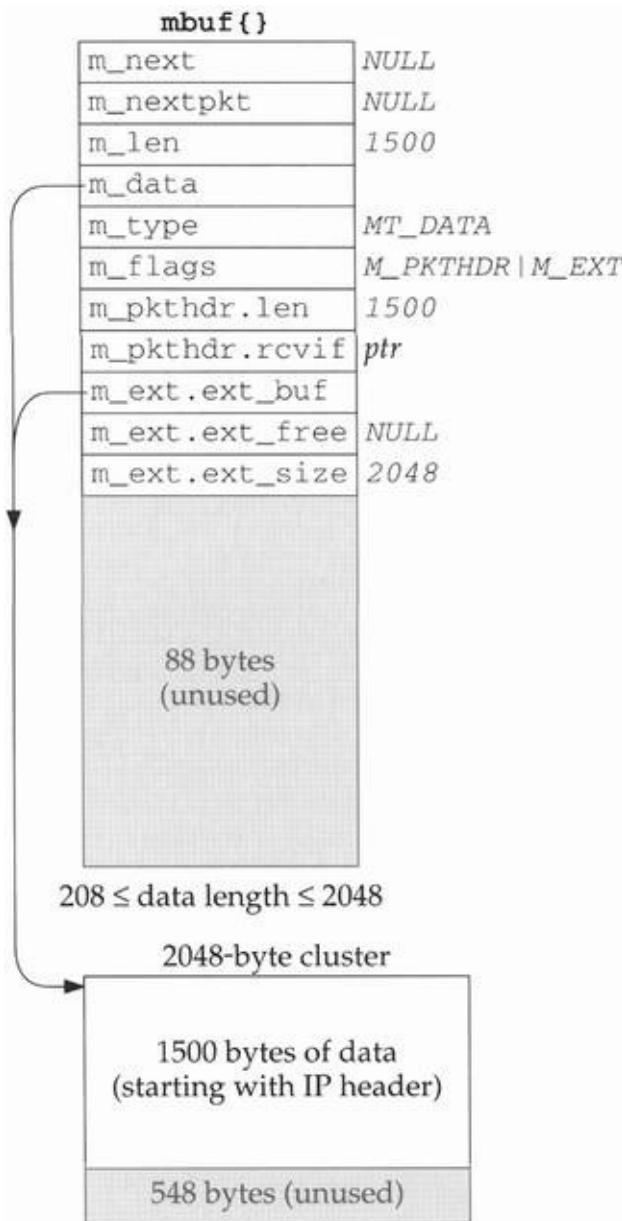
- If the amount of data is between 85 and 100 bytes, there is enough room for the 16 byte header mbuf, but there is no room for the 16 bytes of data. The first mbuf starts at the beginning of the m\_pktdat array and contains the 16 bytes of data. The second mbuf on the right in [Figure 2.15](#) contains the remaining 85 bytes of data.
- [Figure 2.15](#) shows the third type of mbuf creation required when the amount of data is between 100 and 128 bytes. In this case, 16 bytes are stored in the first mbuf (the one with the IP header), and the remaining 85 bytes are stored in the second mbuf. In this case, the m\_pktdat array contains 16 bytes of data, followed by the 85 bytes of data.

**Figure 2.15. Third type of mbuf creation.**



- Figure 2.16 shows the fourth type of mbuf created when the sum of data lengths of all mbufs in a chain is greater than or equal to 208 (MINCLE). In this case, the mbufs are used. The example in the figure assumes a 150 byte clusters. If 1024-byte clusters are in use, then there will be four mbufs, each with the M\_EXT flag set, and each cluster will contain 104 bytes of data.

**Figure 2.16. Fourth type of mbuf**



## mtod and dtom Macros

The two macros mtod and dtom are also define mbuf structure expressions.

```
#define mtod(m, t) ((t)((m)->m_c
```

```
#define dtom(x) ((struct mbuf
```

mtod ("mbuf-to-data") returns a pointer to the casts the pointer to a specified type. For example:

```
struct mbuf *m;
struct ip *ip;

ip = mtod(m, struct ip *);
ip->ip_v = IPVERSION;
```

stores in ip the data pointer of the mbuf (m\_data). The C compiler and the code then references the variable ip. This macro is used when a C structure (often mbuf) is stored in a cluster (Figure 2.15) or if the data is stored in a cluster (Figure 2.16).

The macro dtom ("data-to-mbuf") takes a pointer to the data portion of the mbuf and returns a pointer to the mbuf. For example, if we know that ip points within the data portion of the mbuf:

```
struct mbuf *m;
struct ip *ip;

m = dtom(ip);
```

stores the pointer to the beginning of the mbuf.

is a power of 2, and that mbufs are always aligned by the allocator on MSIZE byte blocks of memory, dtom looks at the order bits in its argument pointer to find the beginning of the mbuf.

There is a problem with dtom: it doesn't work if the mbuf is within a cluster, as in [Figure 2.16](#). Since there is no pointer to the mbuf structure, dtom cannot be used. Therefore, we need to use m\_pullup.

## m\_pullup Function and Contiguous Protocol Headers

The m\_pullup function has two purposes. The first is to pull up a header (e.g., ICMP, IGMP, UDP, or TCP) finds that the amount of data available is less than the size of the minimum protocol header (e.g., 20 bytes for TCP). m\_pullup is called on the assumption that the header is in the next mbuf on the chain. m\_pullup checks that the first  $N$  bytes of data are contiguous in the mbuf and passes the argument to the function that must be less than  $N$ . If the first  $N$  bytes are contiguous in the first mbuf, then dtom will work.

For example, we'll encounter the following code:

```
if (m->m_len < sizeof(struct ip))
    (m = m_pullup(m, sizeof(struct ip)));
    ipstat.ips_toosmall++;
    goto next;
```

```
    }  
    ip = mtod(m, struct ip *);
```

If the amount of data in the first mbuf is less than the header), `m_pullup` is called. `m_pullup` can fail another mbuf and its call to `MGET` fails, or (2) if the mbuf chain is less than the requested number of bytes ( $N$ , which in this case is 20). The second reason for failure. In this example, if `m_pullup` fails, an IP datagram is discarded. Notice that this code assumes the amount of data in the mbuf chain is less than

In actuality, `m_pullup` is rarely called in this scenario. It only calls it when the mbuf length is smaller than the header size, which it normally fails. The reason can be seen by looking at Figure 2.16: there is room in the first mbuf, or in the second mbuf, for 20 bytes, starting with the IP header. This allows for 20 bytes followed by 40 bytes of TCP header. (The UDP header is 8 bytes, and the TCP header is 20 bytes.) If the mbuf chain has more than 60 bytes, then the required number of bytes should always be available. But if the received packet is too short (`m_len` is less than 60), then `m_pullup` is called and it returns an error, because the required number of bytes is not available in the mbuf chain.

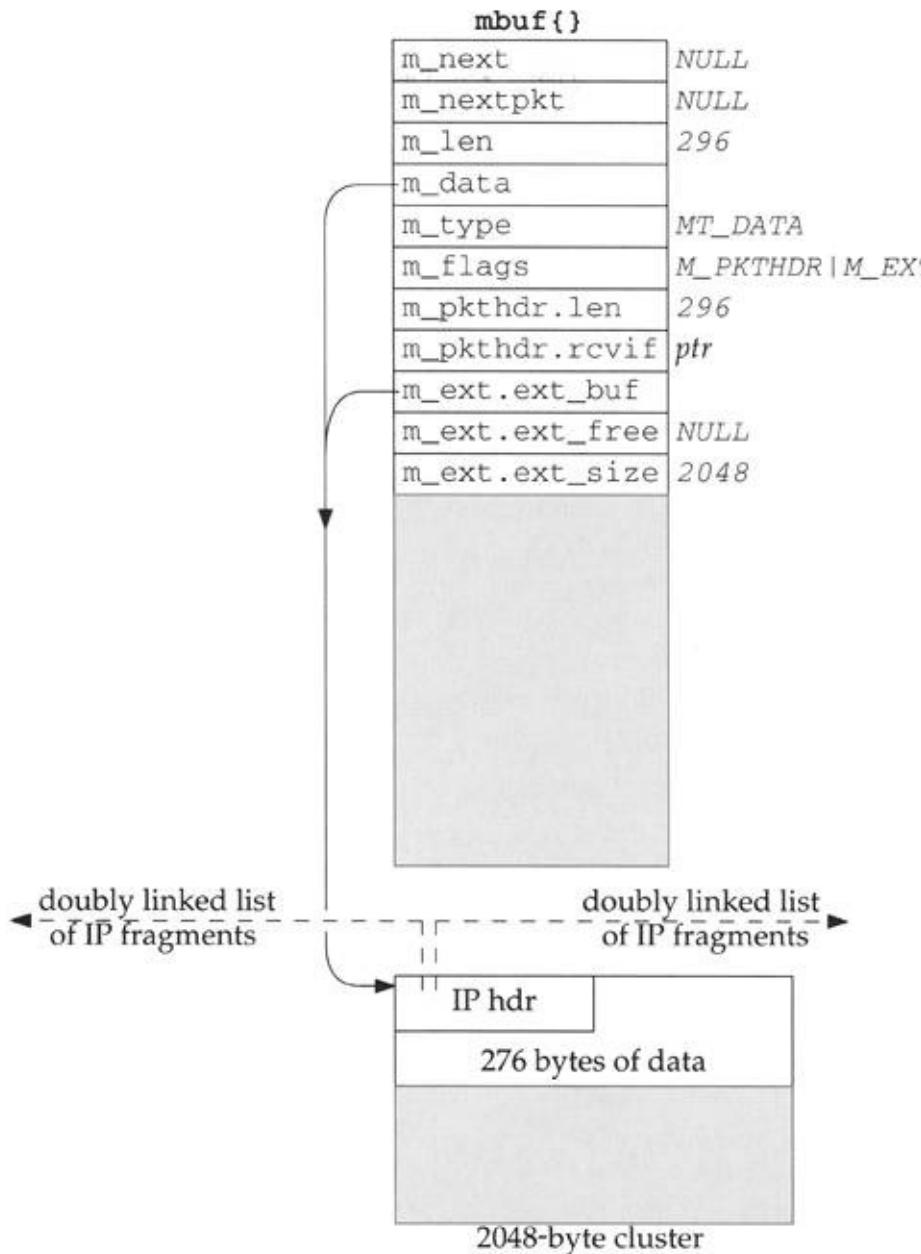
Berkeley-derived kernels maintain a variable `ipstats.mpfail` that is incremented each time `m_pullup` fails. On a Net/3 system I measured, MPFail was 9. The counter ipstats.mpfail is not available in the mbuf chain.

the other protocol counters (i.e., ICMP, IGMP, of m\_pullup were 0. This confirms our statement are because the received IP datagram was too

## **m\_pullup and IP Fragmentation and Reassembly**

The second use of m\_pullup concerns IP reassembly. IP receives a packet of length 296, which is a fragment. The mbuf passed from the device driver to IP is shown in Figure 2.16: the 296 bytes of data are stored in Figure 2.17.

**Figure 2.17. An IP fragment**



The problem is that the IP fragmentation algorithm uses a doubly linked list, using the source and destination header to hold the forward and backward list pointers. These pointers are saved, of course, in the head of the list, since they are needed for reassembled datagram. We describe this in Chapter 2, as shown in [Figure 2.17](#), these linked

and when the list is traversed at some later time (the pointer to the beginning of the cluster) could point to the mbuf. This is the problem we mentioned earlier: the macro cannot be used if `m_data` points into a cluster. A pointer from the cluster to the mbuf. IP fragmentation routine handles this by moving the byte IP header into its own mbuf. The code looks like this:

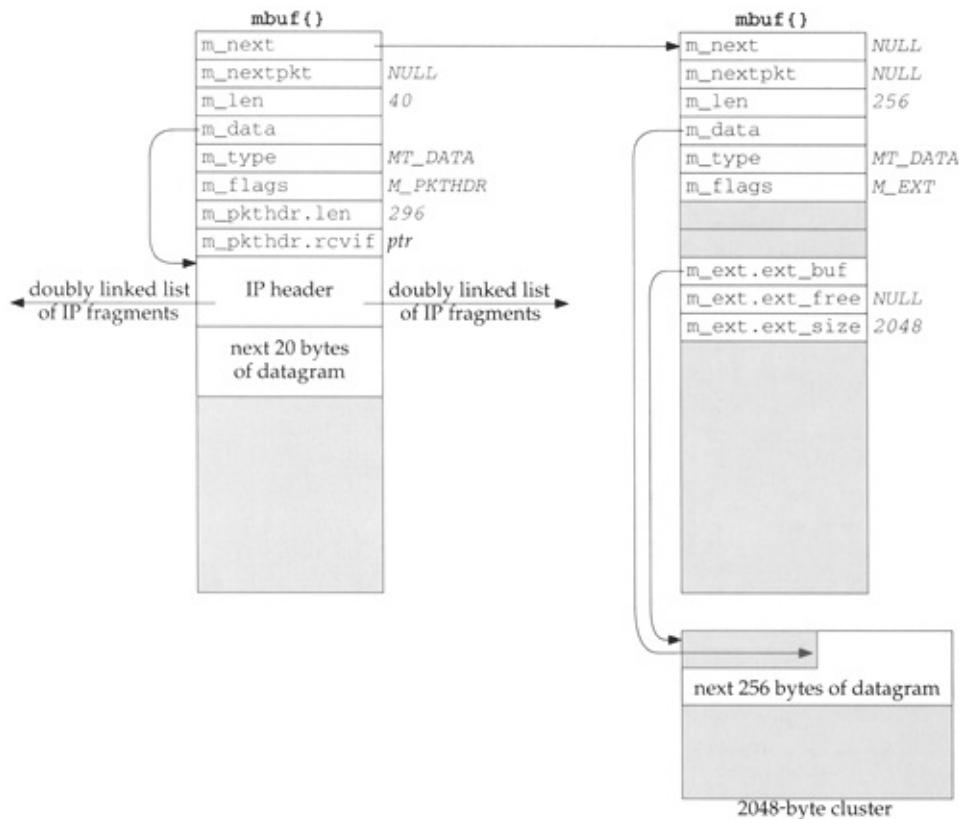
```
if (m->m_flags & M_EXT) {
    if ((m = m_pullup (m, sizeof
                        ipstat.ips_toosmall++;
    goto next;
}
ip = mtod(m, struct ip *);
```

}

[Figure 2.18](#) shows the resulting mbuf chain, after the routine has allocated a new mbuf, prepended it to the chain, and moved the data from the cluster into the new mbuf. The reason for just the requested 20 bytes, is to try to save an additional mbuf. The magic number 40 (max\_protohdr in [Figure 2.19](#)) is the size of the TCP protocol header normally encountered in the cluster, plus a 20-byte TCP header. (This assumes that the cluster contains no other protocols.)

OSI protocols, are not compiled into the kernel

**Figure 2.18. An IP fragment of length**



In Figure 2.18 the IP fragmentation algorithm contained in the mbuf on the left, and this points to the mbuf itself using dtom at a later time.

## Avoidance of m\_pullup by TCP Reassembly

The reassembly of TCP segments uses a different m\_pullup. This is because m\_pullup is expensive

copied from a cluster to an mbuf. TCP tries to avoid this by concatenating the data into a single mbuf whenever possible.

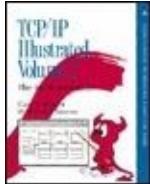
Chapter 19 of Volume 1 mentions that about one-third of the segments contain (often 512 or more bytes of data per segment) the data (of which about 90% of the segments contain the IP header). Hence, when TCP receives segments from IP the data is stored on the left of [Figure 2.14](#) (a small amount of information is contained in the IP header itself) or in the format shown in [Figure 2.16](#) (both the IP header and the TCP data are contained in the same mbuf). If TCP segments arrive out of order, they are stored in the mbufs with IP fragmentation, fields in the IP header are discarded, and the TCP header is accepted which is OK since these fields are no longer needed by TCP. But the same problem arises if the data is copied into the corresponding mbuf pointer, when the pointer is passed to [tcp\\_reass](#) (Figure 2.17).

To solve the problem, we'll see in [Section 27.9](#) how to use some unused fields in the TCP header, providing a back pointer to the mbuf, just to avoid calling `m_pull`. If the IP header is contained in the data portion of the mbuf, this back pointer is superfluous, since the dtom pointer points to the IP header. But if the IP header is contained in a cluster, the back pointer is needed. We'll examine the source code that implements the `m_pullup` and `tcp_reass` functions in [Section 27.9](#).

## Summary of `m_pullup` Usage

We've described three main points about m\_pu

- Most device drivers do not split the first port mbufs. Therefore the possible calls to m\_pul protocol (IP, ICMP, IGMP, UDP, and TCP), just as it is stored contiguously, rarely take place. When it is normally because the IP datagram is too large. If m\_pul returns an error, the datagram is discarded, and the pointer is incremented.
- m\_pullup is called for every received IP fragment stored in a cluster. This means that m\_pullup can return multiple fragments, since the length of most fragments is less than or equal to MCLBYTES.
- As long as TCP segments are not fragmented, m\_pullup can return them in any order. This is one reason to avoid IP fragmentation.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.7 Summary of Mbuf Macros and Functions

Figure 2.19 lists the macros and Figure 2.20 lists the functions that we'll encounter in the code that operates on mbufs. The macros in Figure 2.19 are shown as function prototypes, not as #define statements, to show the data types of the arguments. We will not go through the source code implementation of these routines since they are concerned primarily with manipulating the mbuf data structures and involve no networking issues. Also, there are additional mbuf macros and functions used elsewhere in the Net/3 sources that we don't show in these two figures since we won't encounter

them in the text.

## Figure 2.19. Mbuf macros that we'll encounter in the text.

Macro	Description
MCLGET	Get a cluster (an external buffer) and set the data pointer ( <code>m_data</code> ) of the existing mbuf pointed to by <code>m</code> to point to the cluster. If memory for a cluster is not available, the <code>M_EXT</code> flag in the mbuf is not set on return. <code>void MCLGET(struct mbuf *m, int nowait);</code>
MFREE	Free the single mbuf pointed to by <code>m</code> . If <code>m</code> points to a cluster ( <code>M_EXT</code> is set), the cluster's reference count is decremented but the cluster is not released until its reference count reaches 0 (as discussed in Section 2.9). On return, the pointer to <code>m</code> 's successor (pointed to by <code>m-&gt;m_next</code> , which can be null) is stored in <code>n</code> . <code>void MFREE(struct mbuf *m, struct mbuf *n);</code>
MGETHDR	Allocate an mbuf and initialize it as a packet header. This macro is similar to MGET (Figure 2.12) except the <code>M_PKTHDR</code> flag is set and the data pointer ( <code>m_data</code> ) points to the 100-byte buffer just beyond the packet header. <code>void MGETHDR(struct mbuf *m, int nowait, int type);</code>
MH_ALIGN	Set the <code>m_data</code> pointer of an mbuf containing a packet header to provide room for an object of size <code>len</code> bytes at the end of the mbuf's data area. The data pointer is also longword aligned. <code>void MH_ALIGN(struct mbuf *m, int len);</code>
M_PREFEND	Prepend <code>len</code> bytes of data in front of the data in the mbuf pointed to by <code>m</code> . If room exists in the mbuf, just decrement the pointer ( <code>m_data</code> ) and increment the length ( <code>m_len</code> ) by <code>len</code> bytes. If there is not enough room, a new mbuf is allocated, its <code>m_next</code> pointer is set to <code>m</code> , a pointer to the new mbuf is stored in <code>m</code> , and the data pointer of the new mbuf is set so that the <code>len</code> bytes of data go at the end of the mbuf (i.e., <code>MH_ALIGN</code> is called). Also, if a new mbuf is allocated and the existing mbuf had its packet header flag set, the packet header is moved from the existing mbuf to the new one. <code>void M_PREFEND(struct mbuf *m, int len, int nowait);</code>
dtom	Convert the pointer <code>x</code> , which must point somewhere within the data area of an mbuf, into a pointer to the beginning of the mbuf. <code>struct mbuf *dtom(void *x);</code>
mtod	Type cast the pointer to the data area of the mbuf pointed to by <code>m</code> to <code>type</code> . <code>type mtod(struct mbuf *m, type);</code>

## Figure 2.20. Mbuf functions that we'll encounter in the text.

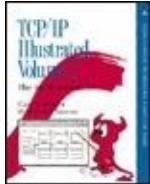
Function	Description
m_adj	Remove <i>len</i> bytes of data from the mbuf chain pointed to by <i>m</i> . If <i>len</i> is positive, that number of bytes is trimmed from the start of the data in the mbuf chain, otherwise the absolute value of <i>len</i> bytes is trimmed from the end of the data in the mbuf chain. <code>void m_adj(struct mbuf *m, int len);</code>
m_cat	Concatenate the mbuf chain pointed to by <i>n</i> to the end of the mbuf chain pointed to by <i>m</i> . We encounter this function when we describe IP reassembly (Chapter 10). <code>void m_cat(struct mbuf *m, struct mbuf *n);</code>
m_copy	A three-argument version of m_copy that implies a fourth argument of M_DONTWAIT. <code>struct mbuf *m_copy(struct mbuf *m, int offset, int len);</code>
m_copydata	Copy <i>len</i> bytes of data from the mbuf chain pointed to by <i>m</i> into the buffer pointed to by <i>cp</i> . The copying starts from the specified byte <i>offset</i> from the beginning of the data in the mbuf chain. <code>void m_copydata(struct mbuf *m, int offset, int len, caddr_t cp);</code>
m_copyback	Copy <i>len</i> bytes of data from the buffer pointed to by <i>cp</i> into the mbuf chain pointed to by <i>m</i> . The data is stored starting at the specified byte <i>offset</i> in the mbuf chain. The mbuf chain is extended with additional mbufs if necessary. <code>void m_copyback(struct mbuf *m, int offset, int len, caddr_t cp);</code>
m_copym	Create a new mbuf chain and copy <i>len</i> bytes of data starting at <i>offset</i> from the mbuf chain pointed to by <i>m</i> . A pointer to the new mbuf chain is returned as the value of the function. If <i>len</i> equals the constant M_COPYALL, the remainder of the mbuf chain starting at <i>offset</i> is copied. We say more about this function in Section 2.9. <code>struct mbuf *m_copym(struct mbuf *m, int offset, int len, int nowait);</code>
m_devget	Create a new mbuf chain with a packet header and return the pointer to the chain. The <i>len</i> and <i>revif</i> fields in the packet header are set to <i>len</i> and <i>ifp</i> . The function <i>copy</i> is called to copy the data from the device interface (pointed to by <i>buf</i> ) into the mbuf. If <i>copy</i> is a null pointer, the function <i>bcopy</i> is called. <i>off</i> is 0 since trailer protocols are no longer supported. We described this function in Section 2.6. <code>struct mbuf *m_devget(char *buf, int len, int off, struct ifnet *ifp, void (*copy)(const void *, void *, u_int));</code>
m_free	A function version of the macro MFREE. <code>struct mbuf *m_free(struct mbuf *m);</code>
m_freem	Free all the mbufs in the chain pointed to by <i>m</i> . <code>void m_freem(struct mbuf *m);</code>
m_get	A function version of the MGET macro. We showed this function in Figure 2.12. <code>struct mbuf *m_get(int nowait, int type);</code>
m_getclr	This function calls the MGET macro to get an mbuf and then zeros the 108-byte buffer. <code>struct mbuf *m_getclr(int nowait, int type);</code>
m_gethdr	A function version of the MGETHDR macro. <code>struct mbuf *m_gethdr(int nowait, int type);</code>
m_pullup	Rearrange the existing data in the mbuf chain pointed to by <i>m</i> so that the first <i>len</i> bytes of data are stored contiguously in the first mbuf in the chain. If this function succeeds, then the <i>mtod</i> macro returns a pointer that correctly references a structure of size <i>len</i> . We described this function in Section 2.6. <code>struct mbuf *m_pullup(struct mbuf *m, int len);</code>

In all the prototypes the argument *nowait* is either M\_WAIT or M\_DONTWAIT, and the argument *type* is one of the MT\_XXX constants shown in Figure 2.10.

As an example of M\_PREPEND, this macro

was called when the IP and UDP headers were prepended to the user's data in the transition from [Figure 1.7](#) to [Figure 1.8](#), causing another mbuf to be allocated. But when this macro was called again (in the transition from [Figure 1.8](#) to [Figure 2.2](#)) to prepend the Ethernet header, room already existed in the mbuf for the headers.

The data type of the last argument for `m_copydata` is `caddr_t`, which stands for "core address." This data type is normally defined in `<sys/types.h>` to be a `char *`. It was originally used internally by the kernel, but got externalized when used by certain system calls. For example, the `mmap` system call, in both 4.4BSD and SVR4, uses `caddr_t` as the type of the first argument and as the return value type.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

## 2.8 Summary of Net/3 Networking Data Structures

This section summarizes the types of data structures we'll encounter in the Net/3 networking code. Other data structures are used in the Net/3 kernel (interested readers should examine the `<sys/queue.h>` header), but the following are the ones we'll encounter in this text.

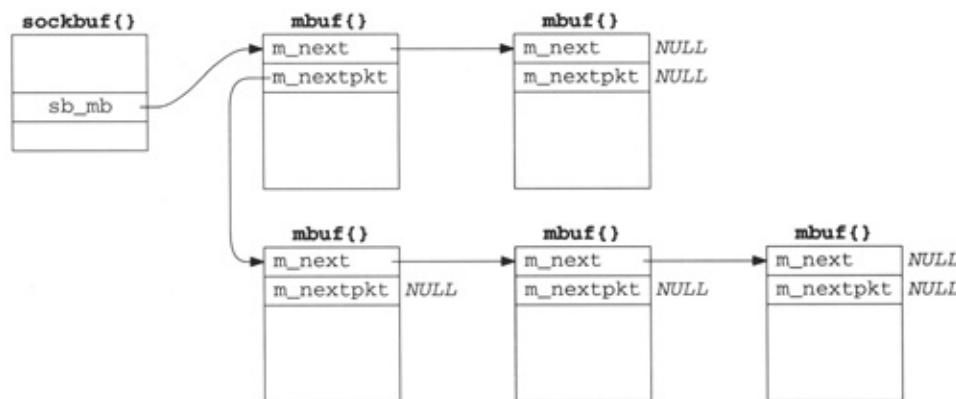
- 1. An mbuf chain: a list of mbufs, linked through the `m_next` pointer. We've seen numerous examples of these already.**

- A linked list of mbuf chains with a head pointer only. The mbuf chains are linked

using the `m_nextpkt` pointer in the first mbuf of each chain.

[Figure 2.21](#) shows this type of list. Examples of this data structure are a socket's send buffer and receive buffer.

## Figure 2.21. Linked list of mbuf chains with head pointer only.



The top two mbufs form the first record on the queue, and the three mbufs on the bottom form the second record on the queue. For a record-based protocol, such as UDP, we can encounter multiple records per queue, but for a protocol such as TCP that has no record boundaries, we'll find only a single record (one mbuf chain possibly consisting of multiple mbufs) per

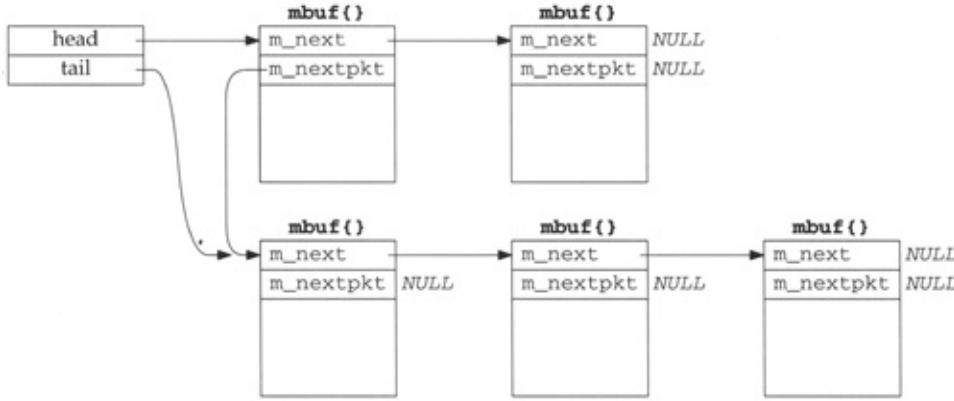
queue.

To append an mbuf to the first record on the queue requires going through all the mbufs comprising the first record, until the one with a null m\_next pointer is encountered. To append an mbuf chain comprising a new record to the queue requires going through all the records until the one with a null m\_nextpkt pointer is encountered.

- A linked list of mbuf chains with head and tail pointers.

[Figure 2.22](#) shows this type of list. We encounter this with the interface queues ([Figure 3.13](#)), and showed an earlier example in [Figure 2.2](#).

**Figure 2.22. Linked list with head and tail pointers.**

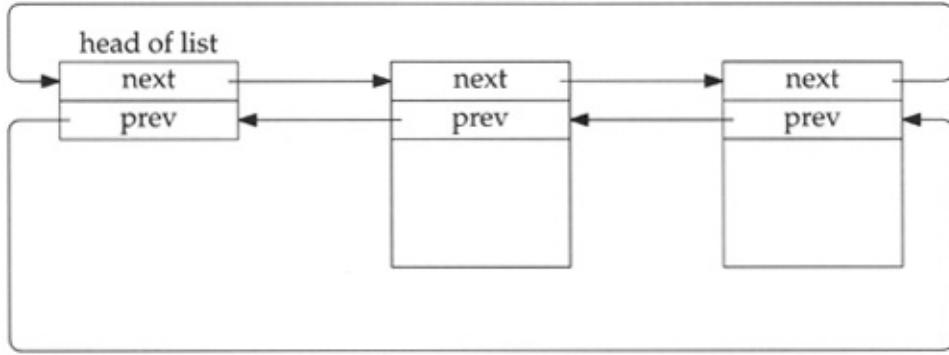


The only change in this figure from [Figure 2.21](#) is the addition of a tail pointer, to simplify the addition of new records.

- A doubly linked, circular list.

[Figure 2.23](#) shows this type of list, which we encounter with IP fragmentation and reassembly ([Chapter 10](#)), protocol control blocks ([Chapter 22](#)), and TCP's out-of-order segment queue ([Section 27.9](#)).

**Figure 2.23. Doubly linked, circular list.**



The elements in the list are not mbufs; they are structures of some type that are defined with two consecutive pointers: a next pointer followed by a previous pointer. Both pointers must appear at the beginning of the structure. If the list is empty, both the next and previous pointers of the head entry point to the head entry.

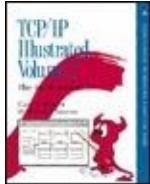
For simplicity in the figure we show the back pointers pointing at another back pointer. Obviously all the pointers contain the address of the structure pointed to, that is the address of a forward pointer (since the forward and backward pointer are always at the beginning of the structure).

This type of data structure allows easy traversal either forward or backward, and

allows easy insertion or deletion at any point in the list.

The functions `insque` and `remque` ([Figure 10.20](#)) are called to insert and delete elements in the list.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.9 m\_copy and Cluster Reference Counts

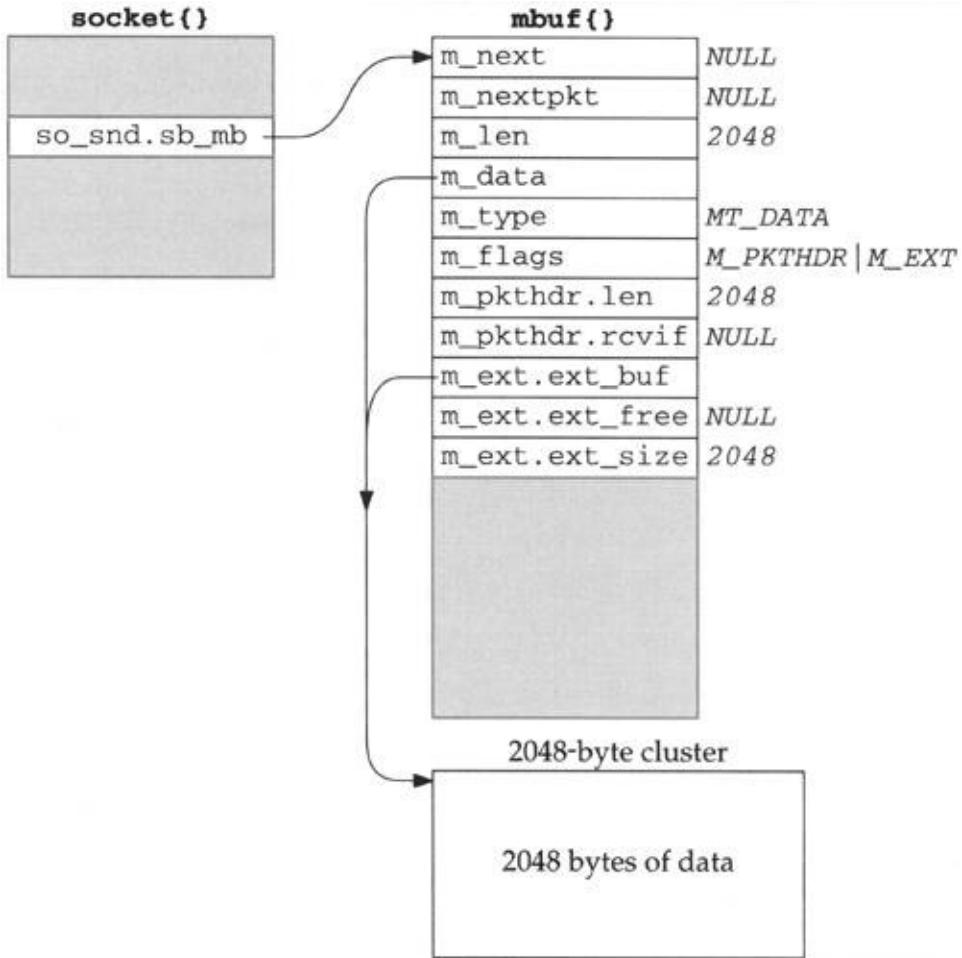
One obvious advantage with clusters is being able to reduce the number of mbufs required to contain large amounts of data. For example, if clusters were not used, it would require 10 mbufs to contain 1024 bytes of data: the first one with 100 bytes of data, the next eight with 108 bytes of data each, and the final one with 60 bytes of data. There is more overhead involved in allocating and linking 10 mbufs, than there is in allocating a single mbuf containing the 1024 bytes in a cluster. A disadvantage with clusters is the potential for wasted space. In our example it takes 2176 bytes using a cluster ( $2048 + 128$ ),

versus 1280 bytes without a cluster ( $10 \times 128$ ).

An additional advantage with clusters is being able to share a cluster between multiple mbufs. We encounter this with TCP output and the `m_copy` function, but describe it in more detail now.

As an example, assume the application performs a write of 4096 bytes to a TCP socket. Assuming the socket's send buffer was previously empty, and that the receiver's window is at least 4096, the following operations take place. One cluster is filled with the first 2048 bytes by the socket layer and the protocol's send routine is called. The TCP send routine appends the mbuf to its send buffer, as shown in [Figure 2.24](#), and calls `tcp_output`.

**Figure 2.24. TCP socket send buffer containing 2048 bytes of data.**

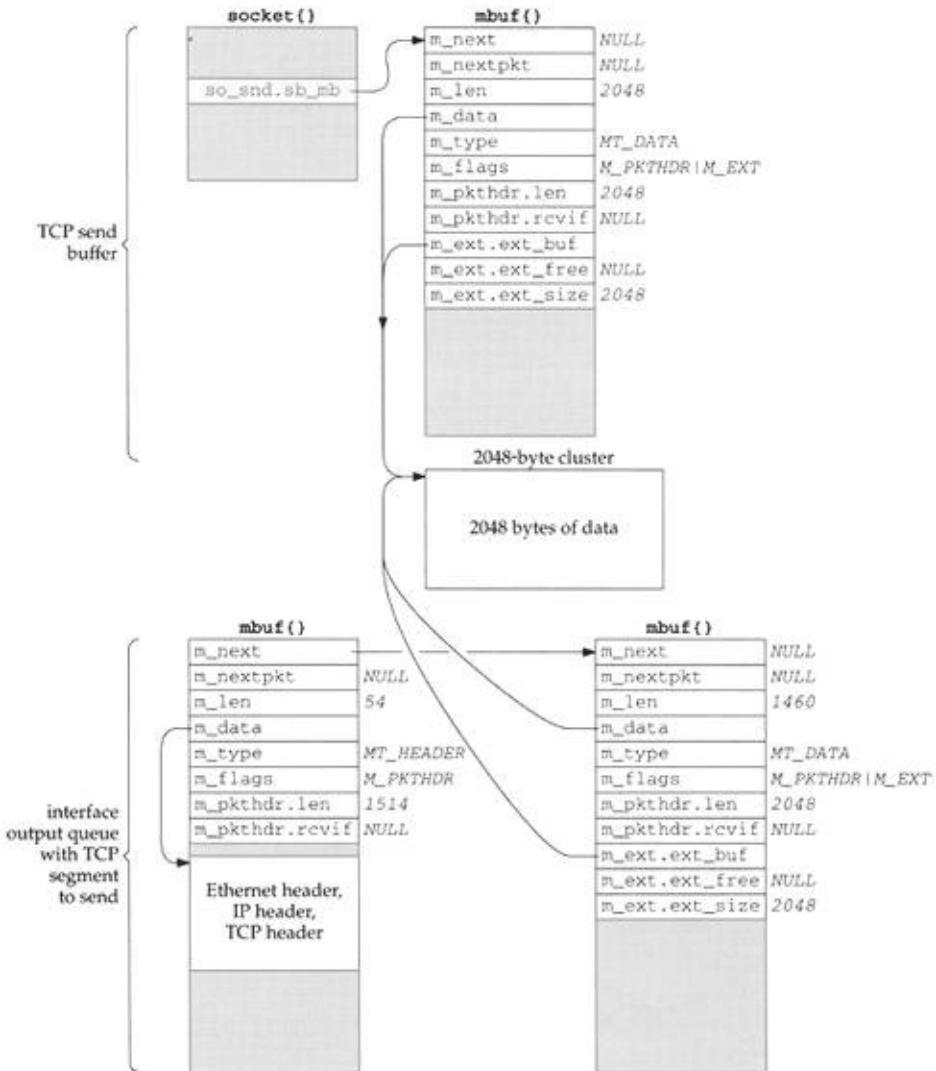


The socket structure contains the sockbuf structure, which holds the head of the list of mbufs in the send buffer:  
`so_snd.sb_mb`.

Assuming a TCP maximum segment size (MSS) of 1460 for this connection (typical for an Ethernet), `tcp_output` builds a segment to send containing the first 1460 bytes of data. It also builds an mbuf containing the IP and TCP headers, leaves

room for a link-layer header (16 bytes), and passes this mbuf chain to IP output. The mbuf chain ends up on the interface's output queue, which we show in [Figure 2.25](#).

**Figure 2.25. TCP socket send buffer and resulting segment on interface's output queue.**



In our UDP example in [Section 1.9](#), UDP took the mbuf chain containing the datagram, prepended an mbuf for the protocol headers, and passed the chain to IP output. UDP did not keep the mbuf in its send buffer. TCP cannot do this since TCP is a reliable protocol and it must maintain a *copy* of the data that it sends, until the data is acknowledged by the other end.

In this example `tcp_output` calls the function `m_copy`, requesting a copy be made of 1460 bytes, starting at offset 0 from the start of its send buffer. But since the data is in a cluster, `m_copy` creates an mbuf (the one on the lower right of [Figure 2.25](#)) and initializes it to point to the correct place in the existing cluster (the beginning of the cluster in this example). The length of this mbuf is 1460, even though an additional 588 bytes of data are in the cluster. We show the length of the mbuf chain as 1514, accounting for the Ethernet, IP, and TCP headers.

We also show this mbuf on the lower right of [Figure 2.25](#) containing a packet header, yet this isn't the first mbuf in the chain. When `m_copy` makes a copy of an mbuf that contains a packet header and the copy starts from offset 0 in the original mbuf, the packet header is also copied verbatim. Since this mbuf is not the first mbuf in the chain, this extraneous packet header is just ignored. The `m_pkthdr.len` value of 2048 in this extraneous packet header is also ignored.

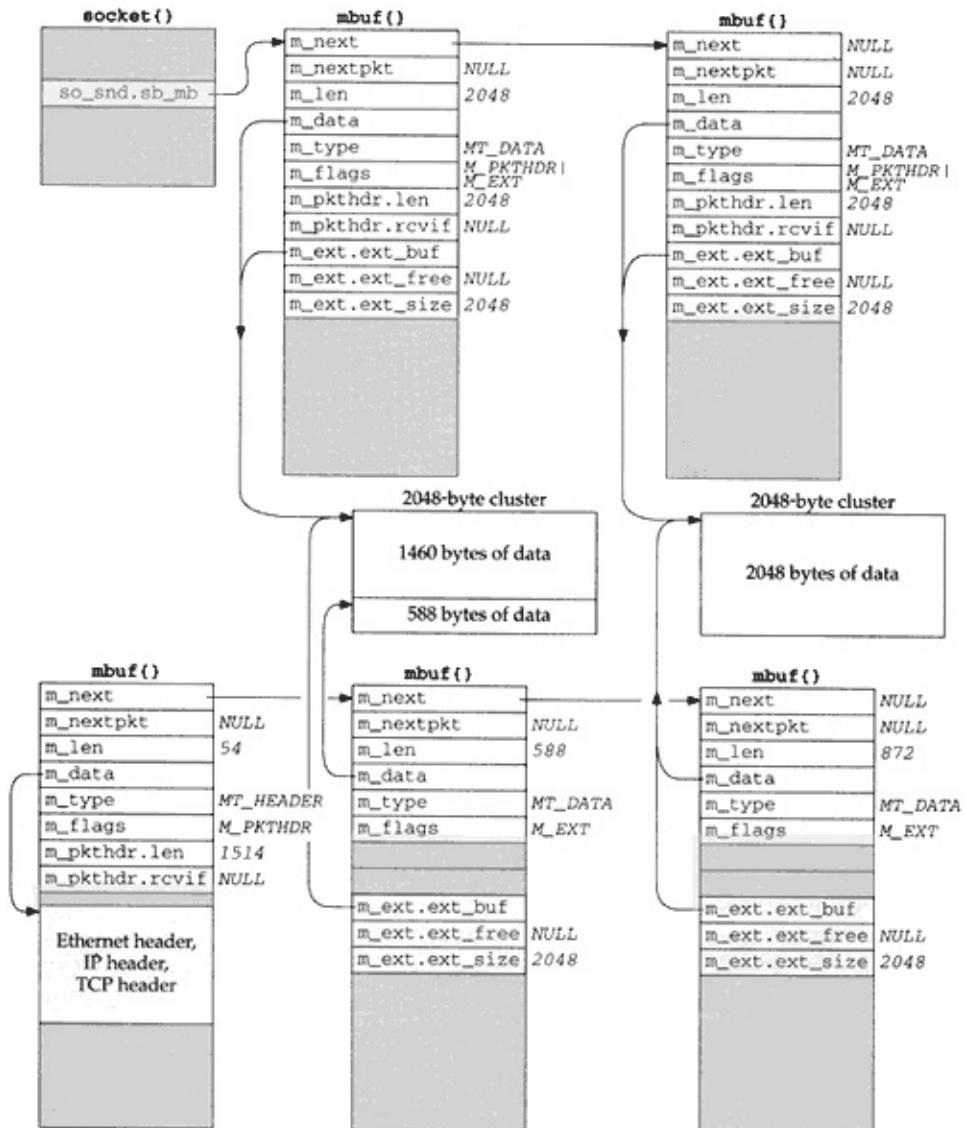
This sharing of clusters prevents the kernel from copying the data from one mbuf into another a big savings. It is implemented by providing a reference count for each cluster that is incremented each time another mbuf points to the cluster, and decremented each time a cluster is released. Only when the reference count reaches 0 is the memory used by the cluster available for some other use. (See [Exercise 2.4](#).)

For example, when the bottom mbuf chain in [Figure 2.25](#) reaches the Ethernet device driver and its contents have been copied to the device, the driver calls `m_freem`. This function releases the first mbuf with the protocol headers and then notices that the second mbuf in the chain points to a cluster. The cluster reference count is decremented, but since its value becomes 1, it is left alone. It cannot be released since it is still in the TCP send buffer.

Continuing our example, `tcp_output` returns after passing the 1460-byte segment to IP, since the remaining 588 bytes in the send buffer don't comprise a

full-sized segment. (In [Chapter 26](#) we describe in detail the conditions under which `tcp_output` sends data.) The socket layer continues processing the data from the application: the remaining 2048 bytes are placed into an mbuf with a cluster, TCP's send routine is called again, and this new mbuf is appended to the socket's send buffer. Since a full-sized segment can be sent, `tcp_output` builds another mbuf chain with the protocol headers and the next 1460 bytes of data. The arguments to `m_copy` specify a starting offset of 1460 bytes from the start of the send buffer and a length of 1460 bytes. This is shown in [Figure 2.26](#), assuming the mbuf chain is again on the interface output queue (so the length of the first mbuf in the chain reflects the Ethernet, IP, and TCP headers).

**Figure 2.26. Mbuf chain to send next 1460-byte TCP segment.**



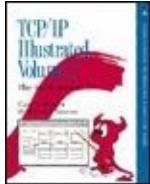
This time the 1460 bytes of data come from two clusters: the first 588 bytes are from the first cluster in the send buffer and the next 872 bytes are from the second cluster in the send buffer. It takes two mbufs to describe these 1460 bytes, but again `m_copy` does not copy the 1460 bytes of data; it references the existing

clusters.

This time we do not show a packet header with either of the mbufs on the bottom right of [Figure 2.26](#). The reason is that the starting offset in the call to `m_copy` is nonzero. Also, we show the second mbuf in the socket send buffer containing a packet header, even though it is not the first mbuf in the chain. This is a property of the `sosend` function, and this extraneous packet header is just ignored.

We encounter the `m_copy` function about a dozen times throughout the text. Although the name implies that a physical copy is made of the data, if the data is contained in a cluster, an additional reference is made to the cluster instead.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.10 Alternatives

Mbufs are far from perfect and they are berated regularly. Nevertheless, they form the basis for all the Berkeley-derived networking code in use today.

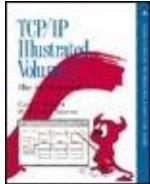
A research implementation of the Internet protocols by Van Jacobson [[Partridge 1993](#)] has done away with the complex mbuf data structures in favor of large contiguous buffers. [[Jacobson 1993](#)] claims a speed improvement of one to two orders of magnitude, although many other changes were made besides getting rid of mbufs.

The complexity of mbufs is a tradeoff that

avoids allocating large fixed buffers that are rarely filled to capacity. At the time mbufs were being designed, a VAX-11/780 with 4 megabytes of memory was a big system, and memory was an expensive resource that needed to be carefully allocated. Today memory is inexpensive, and the focus has shifted toward higher performance and simplicity of code.

The performance of mbufs is also dependent on the amount of data stored in the mbuf. [Hutchinson and Peterson 1991] show that the amount of time required for mbuf processing is nonlinear with respect to the amount of data.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 2. Mbufs: Memory Buffers

### 2.11 Summary

We'll encounter mbufs in almost every function in the text. Their main purpose is to hold the user data that travels from the process to the network interface, and vice versa, but mbufs are also used to contain a variety of other miscellaneous data: source and destination addresses, socket options, and so on.

There are four types of mbufs, depending whether the `M_PKTHDR` and `M_EXT` flags are on or off:

- no packet header, with 0 to 108 bytes of data in mbuf itself,
- packet header, with 0 to 100 bytes of

data in mbuf itself,

- no packet header, with data in cluster (external buffer), and
- packet header, with data in cluster (external buffer).

We looked at the source code for a few of the mbuf macros and functions, but did not present the source code for all the mbuf routines. [Figures 2.19](#) and [2.20](#) provide the function prototypes and descriptions of all the mbuf routines that we encounter in the text.

We looked at the operation of two functions that we'll encounter: `m_devget`, which is called by many network device drivers to store a received frame; and `m_pullup`, which is called by all the input routines to place the required protocol headers into contiguous storage in an mbuf.

The clusters (external buffers) pointed to by an mbuf can be shared by `m_copy`. This is used, for example, by TCP output,

because a copy of the data being transmitted must be maintained by the sender until that data is acknowledged by the other end. Sharing clusters through reference counts is a performance improvement over making a physical copy of the data.

## Exercises

In [Figure 2.9](#) the `M_COPYFLAGS` value **2.1** was defined. Why was the `M_EXT` flag not copied?

In [Section 2.6](#) we listed two reasons that `m_pullup` can fail. There are **2.2** really three reasons. Obtain the source code for this function ([Appendix B](#)) and discover the additional reason.

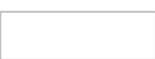
To avoid the problems we described in [Section 2.6](#) with the `dtom` macro when the data is in a cluster, why not just **2.3**

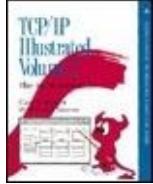
add a back pointer to the mbuf for each cluster?

Since the size of an mbuf cluster is a power of 2 (typically 1024 or 2048), space cannot be taken within the

**2.4** cluster for the reference count. Obtain the Net/3 sources (Appendix B) and determine where these reference counts are stored.

In [Figure 2.5](#) we noted that the two counters `m_drops` and `m_wait` are not **2.5** currently implemented. Modify the mbuf routines to increment these counters when appropriate.





**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 3. Interface Layer

Section 3.1. Introduction

Section 3.2. Code Introduction

Section 3.3. ifnet Structure

Section 3.4. ifaddr Structure

Section 3.5. sockaddr Structure

Section 3.6. ifnet and ifaddr  
Specialization

Section 3.7. Network Initialization  
Overview

Section 3.8. Ethernet Initialization

Section 3.9. SLIP Initialization

Section 3.10. Loopback Initialization

Section 3.11. if\_attach Function

Section 3.12. ifinit Function

## 3.13 Summary

---

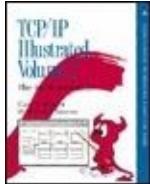
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.1 Introduction

This chapter starts our discussion of Net/3 at the bottom of the protocol stack with the interface layer, which includes the hardware and software that sends and receives packets on locally attached networks.

We use the term *device driver* to refer to the software that communicates with the hardware and *network interface* (or just *interface*) for the hardware and device driver for a particular network.

The Net/3 interface layer attempts to provide a hardware-independent programming interface between the

network protocols and the drivers for the network devices connected to a system. The interface layer provides for all devices:

- a well-defined set of interface functions,
- a standard set of statistics and control flags,
- a device-independent method of storing protocol addresses, and
- a standard queueing method for outgoing packets.

There is no requirement that the interface layer provide reliable delivery of packets, only a best-effort service is required.

Higher protocol layers must compensate for this lack of reliability. This chapter describes the generic data structures maintained for all network interfaces. To illustrate the relevant data structures and algorithms, we refer to three particular network interfaces from Net/3:

- 1. An AMD 7990 LANCE Ethernet interface: an example of a broadcast-capable local area**

## **network.**

- A Serial Line IP (SLIP) interface: an example of a point-to-point network running over asynchronous serial lines.
- A loopback interface: a logical network that returns all outgoing packets as input packets.

---

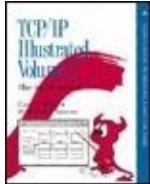
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.2 Code Introduction

The generic interface structures and initialization code are found in three headers and two C files. The device-specific initialization code described in this chapter is found in three different C files. All eight files are listed in [Figure 3.1](#).

**Figure 3.1. Files discussed in this chapter.**

File	Description
sys/socket.h net/if.h net/if_dl.h	address structure definitions interface structure definitions link-level structure definitions
kern/init_main.c net/if.c net/if_loop.c net/if_sl.c hp300/dev/if_le.c	system and interface initialization generic interface code loopback device driver SLIP device driver LANCE Ethernet device driver

## Global Variables

The global variables introduced in this chapter are described in [Figure 3.2.](#)

**Figure 3.2. Global variables introduced in this chapter.**

Variable	Data type	Description
pdevinit	struct pdevinit []	array of initialization parameters for pseudo-devices such as SLIP and loopback interfaces
ifnet	struct ifnet *	head of list of ifnet structures
ifnet_addrs	struct ifaddr **	array of pointers to link-level interface addresses
if_indexlim	int	size of ifnet_addrs array
if_index	int	index of the last configured interface
ifqmaxlen	int	maximum size of interface output queues
hz	int	the clock-tick frequency for this system (ticks/second)

## SNMP Variables

The Net/3 kernel collects a wide variety of networking statistics. In most chapters we summarize the statistics and show how

they relate to the standard TCP/IP information and statistics defined in the Simple Network Management Protocol Management Information Base (SNMP MIB-II). RFC 1213 [[McCloghrie and Rose 1991](#)] describe SNMP MIB-II, which is organized into 10 distinct information groups shown in [Figure 3.3](#).

**Figure 3.3. SNMP groups in MIB-II.**

SNMP Group	Description
System	general information about the system
Interfaces	network interface information
Address Translation	network-address-to-hardware-address-translation tables (deprecated)
IP	IP protocol information
ICMP	ICMP protocol information
TCP	TCP protocol information
UDP	UDP protocol information
EGP	EGP protocol information
Transmission	media-specific information
SNMP	SNMP protocol information

Net/3 does not include an SNMP agent. Instead, an SNMP agent for Net/3 is implemented as a process that accesses the kernel statistics in response to SNMP queries through the mechanism described in [Section 2.2](#).

While most of the MIB-II variables are collected by Net/3 and may be accessed directly by an SNMP agent, others must be derived indirectly. MIB-II variables fall into three categories: (1) simple variables such as an integer value, a timestamp, or a byte string; (2) lists of simple variables such as an individual routing entry or an interface description entry; and (3) lists of lists such as the entire routing table and the list of all interface entries.

The ISO/DE package includes a sample SNMP agent for Net/3. See [Appendix B](#) for information about ISO/DE.

[Figure 3.4](#) shows the one simple variable maintained for the SNMP interface group. We describe the SNMP interface table later in [Figure 4.7](#).

### Figure 3.4. Simple SNMP variable in the interface group.

SNMP variable	Net/3 variable	Description
ifNumber	if_index + 1	if_index is the index of the last interface in the system and starts at 0; 1 is added to get ifNumber, the number of interfaces in the system.

---

**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

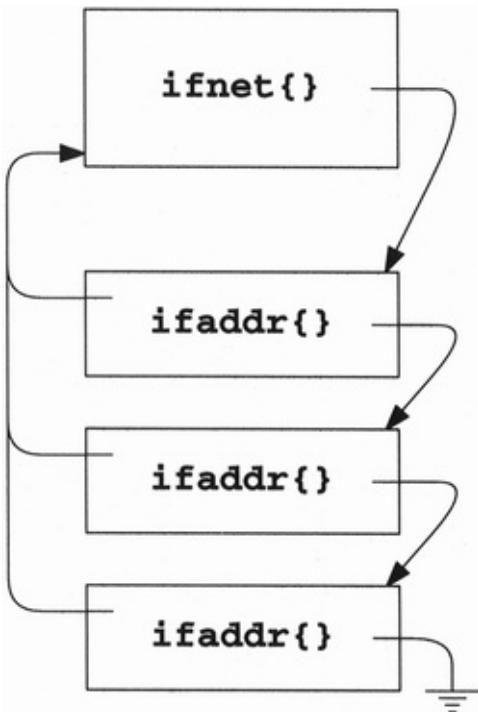
[Top](#)

## Chapter 3. Interface Layer

### 3.3 ifnet Structure

The ifnet structure contains information common to all interfaces. During initialization, a separate ifnet structure is allocated for each interface. This structure has a list of one or more protocol addresses associated with it. Figure 3.5 illustrates the relationship between an interface and its ifnet structure.

**Figure 3.5. Each ifnet structure has a list of one or more protocol addresses.**



The interface in Figure 3.5 is shown with three structures. Although some network interfaces, protocol, others, such as Ethernet, support multiple addresses. For example, a system may use a single interface to support both Internet and OSI protocols. A type field identifies the protocol. The Internet and OSI protocols employ different address formats. An interface must have an Internet address and an OSI address. The three ifaddr structures are connected by a linked list (the arrows on the right). Each ifaddr structure has a back pointer to the related ifnet structure (the arrow on the left).

It is also possible for a single network interface to support multiple addresses for a single protocol. For example, two Internet addresses for a single Ethernet interface in Net/3.

This feature first appeared in Net/2. Having two or more addresses for a single interface can be useful when renumbering a network. During a transition period,

packets addressed to the old and new addresses.

The ifnet structure is large so we describe it in

- implementation information,
- hardware information,
- interface statistics,
- function pointers, and
- the output queue.

Figure 3.6 shows the implementation information.

**Figure 3.6. ifnet structure: implementation**

```
-----if.h-----  
80 struct ifnet {  
81     struct ifnet *if_next;      /* all struct ifnets are chained */  
82     struct ifaddr *if_addrlist; /* linked list of addresses per if */  
83     char *if_name;           /* name, e.g. 'le' or 'lo' */  
84     short if_unit;          /* sub-unit for lower level driver */  
85     u_short if_index;        /* numeric abbreviation for this if */  
86     short if_flags;          /* Figure 3.7 */  
87     short if_timer;          /* time 'til if_watchdog called */  
88     int if_pcount;           /* number of promiscuous listeners */  
89     caddr_t if_bpf;          /* packet filter structure */  
-----if.h-----
```

80-82

if\_next joins the ifnet structures for all the interfaces.  
The if\_init function constructs the list during system initialization.

ifaddr structures for the interface ([Figure 3.16](#)) information for a protocol that expects to comm

## Common interface information

83-86

if\_name is a short string that identifies the interface instances of the same type. For example, if a set of interfaces have an if\_name consisting of the 2 bytes "s1" and 1 for the second. if\_index uniquely identifies the interface used by the sysctl system call ([Section 19.14](#)) :  
The if\_index field is used to identify the interface.

Sometimes an interface is not uniquely identified. For example, several SLIP connections can have the same if\_name. In this case, the user specifies the interface explicitly.

if\_flags specifies the operational state and properties of the interface. The flags are examined but cannot change the flag value. For example, the flags are accessed with the SIOCGIFFLAGS command described in [Section 4.4](#).

**Figure 3.7. if\_flag**

<code>if_flags</code>	Kernel only	Description
<code>IFF_BROADCAST</code> <code>IFF_MULTICAST</code> <code>IFF_POINTOPOINT</code> <code>IFF_LOOPBACK</code> <code>IFF_OACTIVE</code> <code>IFF_RUNNING</code> <code>IFF_SIMPLEX</code>	• • • • • • •	the interface is for a broadcast network the interface supports multicasting the interface is for a point-to-point network the interface is for a loopback network a transmission is in progress resources are allocated for this interface the interface cannot receive its own transmissions
<code>IFF_LINK0</code> <code>IFF_LINK1</code> <code>IFF_LINK2</code>	see text see text see text	defined by device driver defined by device driver defined by device driver
<code>IFF_ALLMULTI</code> <code>IFF_DEBUG</code> <code>IFF_NOARP</code> <code>IFF_NOTRAILERS</code> <code>IFF_PROMISC</code> <code>IFF_UP</code>		the interface is receiving all multicast packets debugging is enabled for the interface don't use ARP on this interface avoid using trailer encapsulation the interface receives all network packets the interface is operating

## The IFF\_BROADCAST and IFF\_POINTOPOINT

The macro `IFF_CANTCHANGE` is a bitwise OR column.

The device-specific flags (`IFF_LINKx`) may or depending on the device. For example, [Figure](#) by the SLIP driver.

## Interface timer

87

`if_timer` is the time in seconds until the kernel re-enables the interface. This function may be used by the device driver to poll the hardware at regular intervals or to reset hardware that isn't responding.

## BSD Packet Filter

88-89

The next two members, if\_pcount and if\_bpf, are used to support packet filtering. Through BPF, a process can receive copies of packets transmitted or received by an interface. As we discuss the device drivers, we will see how BPF is used. BPF itself is described in [Chapter 31](#).

The next section of the ifnet structure, shown in Figure 3.8, describes the characteristics of the interface.

**Figure 3.8. ifnet structure: interface characteristics**

```
if.h
90 struct if_data {
91 /* generic interface information */
92     u_char ifi_type;      /* Figure 3.9 */
93     u_char ifi_addrlen;   /* media address length */
94     u_char ifi_hdrlen;    /* media header length */
95     u_long ifi_mtu;       /* maximum transmission unit */
96     u_long ifi_metric;    /* routing metric (external only) */
97     u_long ifi_baudrate;  /* linespeed */

         /* other ifnet members */

138 #define if_mtu      if_data_ifi_mtu
139 #define if_type      if_data_ifi_type
140 #define if_addrlen   if_data_ifi_addrlen
141 #define if_hdrlen    if_data_ifi_hdrlen
142 #define if_metric    if_data_ifi_metric
143 #define if_baudrate  if_data_ifi_baudrate
if.h
```

Net/3 and this text use the short names provided in lines 138 through 143 to specify the ifnet members.

## Interface characteristics

90-92

`if_type` specifies the hardware address type supported by several common values from `net/if_types.h`.

**Figure 3.9. `if_type`:**

<code>if_type</code>	Description
<code>IFT_OTHER</code>	unspecified
<code>IFT_ETHER</code>	Ethernet
<code>IFT_ISO88023</code>	IEEE 802.3 Ethernet (CMSA/CD)
<code>IFT_ISO88025</code>	IEEE 802.5 token ring
<code>IFT_FDDI</code>	Fiber Distributed Data Interface
<code>IFT_LOOP</code>	loopback interface
<code>IFT_SLIP</code>	serial line IP

93-94

`if_addrlen` is the length of the datalink address attached to any outgoing packet by the hardware. For Ethernet, it has an address length of 6 bytes and a header length of 14 bytes.

95

`if_mtu` is the maximum transmission unit of the interface. It is the maximum unit of data that the interface can transmit in a single frame. It is an important parameter that controls the size of packets transmitted over the network. For Ethernet, the value is 1500 bytes.

96-97

if\_metric is usually 0; a higher value makes routing prefer this interface.  
if\_baudrate specifies the transmission speed of the interface.

Interface statistics are collected by the next group of fields shown in Figure 3.10.

Figure 3.10. ifnet structure

```
98 /* volatile statistics */                                if.h
99     u_long ifi_ipackets;    /* #packets received on interface */
100    u_long ifi_ierrors;    /* #input errors on interface */
101    u_long ifi_opackets;   /* #packets sent on interface */
102    u_long ifi_oerrors;    /* #output errors on interface */
103    u_long ifi_collisions; /* #collisions on csma interfaces */
104    u_long ifi_ibytes;     /* #bytes received */
105    u_long ifi_obytes;     /* #bytes sent */
106    u_long ifi_imcasts;    /* #packets received via multicast */
107    u_long ifi_omcasts;    /* #packets sent via multicast */
108    u_long ifi_iqdrops;    /* #packets dropped on input, for this
                               interface */
109    u_long ifi_noproto;    /* #packets destined for unsupported
                               protocol */
110
111    struct timeval ifi_lastchange; /* last updated */
112
113 } if_data;
114
115 /* other ifnet members */

144 #define if_ipackets if_data.ifi_ipackets
145 #define if_ierrors if_data.ifi_ierrors
146 #define if_opackets if_data.ifi_opackets
147 #define if_oerrors if_data.ifi_oerrors
148 #define if_collisions if_data.ifi_collisions
149 #define if_ibytes if_data.ifi_ibytes
150 #define if_obytes if_data.ifi_obytes
151 #define if_imcasts if_data.ifi_imcasts
152 #define if_omcasts if_data.ifi_omcasts
153 #define if_iqdrops if_data.ifi_iqdrops
154 #define if_noproto if_data.ifi_noproto
155 #define if_lastchange if_data.ifi_lastchange
```

## Interface statistics

**98-111**

Most of these statistics are self-explanatory. if\_transmission is interrupted by another transmission. if\_noproto counts the number of packets that cannot be supported by the system or the interface (e.g., a system that supports only IP). The SLIP interface places the packet on its output queue.

These statistics were not part of the ifnet structure, but support the standard SNMP MIB-II variables for interfaces.

if\_iqdrops is accessed only by the SLIP device drivers, which increment if\_snd.ifq\_drops (Figure 3.1). This was already in the BSD software when the SNMP agent ignores if\_iqdrops and uses ifsnd.

## Change timestamp

**112-113**

if\_lastchange records the last time any of the statistics changed.

Once again, Net/3 and this text use the short statements on lines 144 through 155 to specify the same thing.

The next section of the ifnet structure, shown in Figure 3.2, contains standard interface-layer functions, which isolate the interface from the rest of the system.

layer. Each network interface implements these device.

**Figure 3.11. ifnet structure**

```
if.h
114 /* procedure handles */
115     int (*if_init)          /* init routine */
116             (int);
117     int (*if_output)        /* output routine (enqueue) */
118             (struct ifnet *, struct mbuf *, struct sockaddr *,
119              struct rtentry *);
120     int (*if_start)         /* initiate output routine */
121             (struct ifnet *);
122     int (*if_done)          /* output complete routine */
123             (struct ifnet *); /* (XXX not used; fake prototype) */
124     int (*if_ioctl)         /* ioctl routine */
125             (struct ifnet *, int, caddr_t);
126     int (*if_reset)
127             (int);           /* new autoconfig will permit removal */
128     int (*if_watchdog)      /* timer routine */
129             (int);
```

## Interface functions

114-129

Each device driver initializes its own ifnet structure at system initialization time. [Figure 3.12](#) describes

**Figure 3.12. ifnet structure**

Function	Description
if_init	initialize the interface
if_output	queue outgoing packets for transmission
if_start	initiate transmission of packets
if_done	cleanup after transmission completes (not used)
if_ioctl	process I/O control commands
if_reset	reset the interface device
if_watchdog	periodic interface routine

We will see the comment /\* XXX \*/ throughout that the code is obscure, contains nonobvious more difficult problem. In this case, it indicates

In [Chapter 4](#) we look at the device-specific function interfaces, which the kernel calls indirectly through, for example, if ifp points to an ifnet structure,

```
(*ifp->if_start) (ifp)
```

calls the if\_start function of the device driver as

The remaining member of the ifnet structure is shown in [Figure 3.13](#).

**Figure 3.13. ifnet structure**

```
if.h
130 struct ifqueue {
131     struct mbuf *ifq_head;
132     struct mbuf *ifq_tail;
133     int     ifq_len;          /* current length of queue */
134     int     ifq_maxlen;       /* maximum length of queue */
135     int     ifq_drops;        /* packets dropped because of full queue */
136 } if_snd;                  /* output queue */
137 };
if.h
```

130-137

if\_snd is the queue of outgoing packets for the structure and therefore its own output queue. ifq\_head points to the head of the queue (the next one to be output), ifq\_tail points to the end of the queue. ifq\_len is the number of packets currently on the queue. ifq\_maxlen is the maximum number of buffers allowed on the queue. This number is controlled by the integer ifqmaxlen, which is initialized at compilation time and can be changed by the driver. The queue is implemented as a linked list of mbuf structures. ifq\_drops counts the number of packets discarded because they could not be queued. The structure is used by macros and functions that access a queue.

Figure 3.14. ifqueue structure

Function	Description
IF_QFULL	Is <i>ifq</i> full?  int <b>IF_QFULL</b> (struct ifqueue * <i>ifq</i> );
IF_DROP	IF_DROP only increments the <i>ifq_drops</i> counter associated with <i>ifq</i> . The name is misleading; the <i>caller</i> drops the packet.  void <b>IF_DROP</b> (struct ifqueue * <i>ifq</i> );
IF_ENQUEUE	Add the packet <i>m</i> to the end of the <i>ifq</i> queue. Packets are linked together by <i>m_nextpkt</i> in the mbuf header.  void <b>IF_ENQUEUE</b> (struct ifqueue * <i>ifq</i> , struct mbuf * <i>m</i> );
IF_PREPEND	Insert the packet <i>m</i> at the front of the <i>ifq</i> queue.  void <b>IF_PREPEND</b> (struct ifqueue * <i>ifq</i> , struct mbuf * <i>m</i> );
IF_DEQUEUE	Take the first packet off the <i>ifq</i> queue. <i>m</i> points to the dequeued packet or is null if the queue was empty.  void <b>IF_DEQUEUE</b> (struct ifqueue * <i>ifq</i> , struct mbuf * <i>m</i> );
if_qflush	Discard all packets on the queue <i>ifq</i> , for example, when an interface is shut down.  void <b>if_qflush</b> (struct ifqueue * <i>ifq</i> );

The first five routines are macros defined in net.h. The last one is a function defined in net/if.c. The macros often appear together.

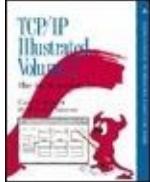
```
s = splimp();
if (IF_QFULL(inq)) {
    IF_DROP(inq); /* queue full */
    m_freem(m);
} else
    IF_ENQUEUE(inq, m); /* there's space */
splx(s);
```

This code fragment attempts to add a packet to a queue that is full. It increments *ifq\_drops* and the packet is discarded. This is how the system handles retransmit discarded packets. Applications using this code must handle the case where *ifq* is full.

detect and handle the retransmission on their c

Access to the queue is bracketed by splimp and prevent the network interrupt service routines from entering an indeterminate state.

m\_freem is called before splx because the mbufs must be freed before the interrupt level is raised. It would be wasted effort to call splx to leave a critical section during m\_freem ([Section 2.5](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.4 ifaddr Structure

The next structure we look at is the interface address structure, ifaddr, shown in [Figure 3.15](#). Each interface maintains a linked list of ifaddr structures because some data links, such as Ethernet, support more than one protocol. A separate ifaddr structure describes each address assigned to the interface, usually one address per protocol. Another reason to support multiple addresses is that many protocols, including TCP/IP, support multiple addresses assigned to a single physical interface. Although Net/3 supports this feature, many implementations of TCP/IP do not.

## Figure 3.15. ifaddr structure.

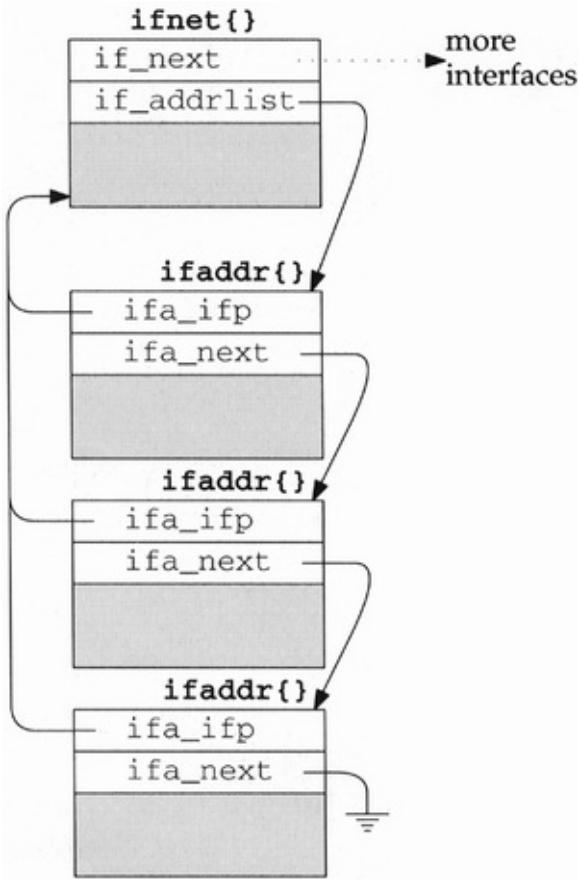
```
if.h
217 struct ifaddr {
218     struct ifaddr *ifa_next;      /* next address for interface */
219     struct ifnet *ifa_ifp;        /* back-pointer to interface */
220     struct sockaddr *ifa_addr;   /* address of interface */
221     struct sockaddr *ifa_dstaddr; /* other end of p-to-p link */
222 #define ifa_broadaddr ifa_dstaddr /* broadcast address interface */
223     struct sockaddr *ifa_netmask; /* used to determine subnet */
224     void (*ifa_rtrequest)();    /* check or clean routes */
225     u_short ifa_flags;          /* mostly rt_flags for cloning */
226     short ifa_refcnt;           /* references to this structure */
227     int ifa_metric;             /* cost for this interface */
228 };
if.h
```

217-219

The ifaddr structure links all addresses assigned to an interface together by ifa\_next and contains a pointer, ifa\_ifp, back to the interface's ifnet structure.

[Figure 3.16](#) shows the relationship between the ifnet structures and the ifaddr structures.

## Figure 3.16. ifnet and ifaddr structures.



220

`ifa_addr` points to a protocol address for the interface and `ifa_netmask` points to a bit mask that selects the network portion of `ifa_addr`. Bits that represent the network portion of the address are set to 1 in the mask, and the host portion of the address is set to all 0 bits. Both addresses are stored as `sockaddr` structures ([Section 3.5](#)). [Figure 3.38](#) shows an address and its related mask structure. For IP addresses, the mask selects the network and subnet

portions of the IP address.

221-223

`ifa_dstaddr` (or its alias `ifa_broadaddr`) points to the protocol address of the interface at the other end of a point-to-point link or to the broadcast address assigned to the interface on a broadcast network such as Ethernet. The mutually exclusive flags `IFF_BROADCAST` and `IFF_POINTOPOINT` ([Figure 3.7](#)) in the interface's `ifnet` structure specify the applicable name.

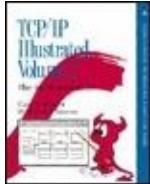
224-228

`ifa_rtrequest`, `ifa_flags`, and `ifa_metric` support routing lookups for the interface.

`ifa_refcnt` counts references to the `ifaddr` structure. The macro `IFAFREE` only releases the structure when the reference count drops to 0, such as when addresses are deleted with the `SIOCDIFADDR` ioctl command. The `ifaddr` structures are reference-counted because they are shared by the interface and routing data

structures.

IFAFREE decrements the counter and returns if there are other references. This is the common case and avoids a function call overhead for all but the last reference. If this is the last reference, IFAFREE calls the function ifafree, which releases the structure.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.5 sockaddr Structure

Addressing information for an interface consists of more than a single host address. Net/3 maintains host, broadcast, and network masks in structures derived from a generic sockaddr structure. By using a generic structure, hardware and protocol-specific addressing details are hidden from the interface layer.

[Figure 3.17](#) shows the current definition of the structure as well as the definition from earlier BSD releases an osockaddr structure.

**Figure 3.17. sockaddr and osockaddr**

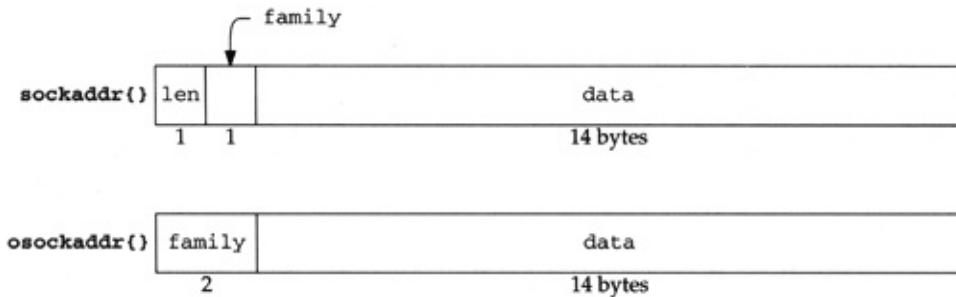
## structures.

```
----- socket.h
120 struct sockaddr {
121     u_char    sa_len;           /* total length */
122     u_char    sa_family;        /* address family (Figure 3.19) */
123     char     sa_data[14];       /* actually longer; address value */
124 };

271 struct osockaddr {
272     u_short   sa_family;        /* address family (Figure 3.19) */
273     char     sa_data[14];       /* up to 14 bytes of direct address */
274 };
----- socket.h
```

Figure 3.18 illustrates the organization of these structures.

**Figure 3.18. sockaddr and osockaddr structures (sa\_ prefix dropped).**



In many figures, we omit the common prefix in member names. In this case, we've dropped the `sa_` prefix.

## sockaddr structure

## 102-124

Every protocol has its own address format. Net/3 handles generic addresses in a sockaddr structure. sa\_len specifies the length of the address (OSI and Unix domain protocols have variable-length addresses) and sa\_family specifies the type of address. [Figure 3.19](#) lists the *address family* constants that we encounter.

**Figure 3.19. sa\_family constants.**

sa_family	Protocol
<i>AF_INET</i>	Internet
<i>AF_ISO, AF_OSI</i>	OSI
<i>AF_UNIX</i>	Unix
<i>AF_ROUTE</i>	routing table
<i>AF_LINK</i>	data link
<i>AF_UNSPEC</i>	(see text)

The contents of a sockaddr when AF\_UNSPEC is specified depends on the context. In most cases, it contains an Ethernet hardware address.

The sa\_len and sa\_family members allow protocol-independent code to manipulate

variable-length sockaddr structures from multiple protocol families. The remaining member, `sa_data`, contains the address in a protocol-dependent format. `sa_data` is defined to be an array of 14 bytes, but when the sockaddr structure overlays a larger area of memory `sa_data` may be up to 253 bytes long. `sa_len` is only a single byte, so the size of the entire address including `sa_len` and `sa_family` must be less than 256 bytes.

This is a common C technique that allows the programmer to consider the last member in a structure to have a variable length.

Each protocol defines a specialized sockaddr structure that duplicates the `sa_len` and `sa_family` members but defines the `sa_data` member as required for that protocol. The address stored in `sa_data` is a transport address; it contains enough information to identify multiple communication end points on the same host. In [Chapter 6](#) we look at the Internet address structure `sockaddr_in`, which consists of an IP address and a port

number.

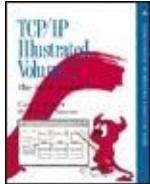
## osockaddr structure

271-274

The osockaddr structure is the definition of a sockaddr before the 4.3BSD Reno release. Since the length of an address was not explicitly available in this definition, it was not possible to write protocol-independent code to handle variable-length addresses. The desire to include the OSI protocols, which utilize variable-length addresses, motivated the change in the sockaddr definition seen in Net/3. The osockaddr structure is supported for binary compatibility with previously compiled programs.

We have omitted the binary compatibility code from this text.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

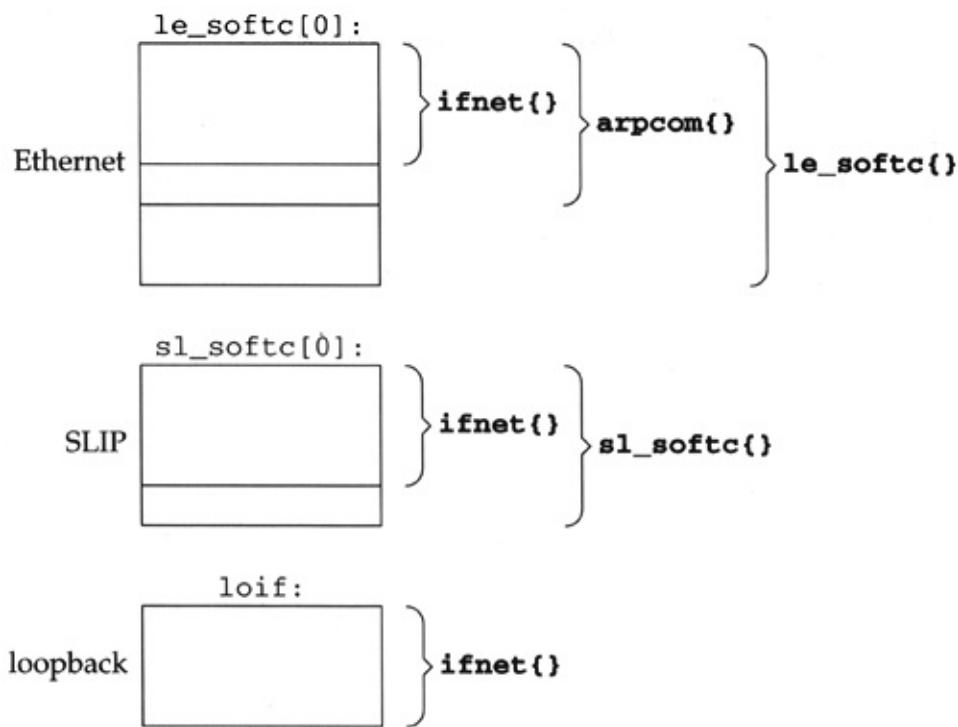
### 3.6 ifnet and ifaddr Specialization

The ifnet and ifaddr structures contain general information applicable to all network interfaces and protocol addresses. To accommodate additional device and protocol-specific information, each driver defines and each protocol allocates a specialized version of the ifnet and ifaddr structures. These specialized structures always contain an ifnet or ifaddr structure as their first member so that the common information can be accessed without consideration for the additional specialized information.

Most device drivers handle multiple

interfaces of the same type by allocating an array of its specialized ifnet structures, but others (such as the loopback driver) handle only one interface. [Figure 3.20](#) shows the arrangement of specialized ifnet structures for our sample interfaces.

**Figure 3.20. Arrangement of ifnet structures within device-dependent structures.**



Notice that each device's structure begins with an ifnet structure, followed by all the

device-dependent data. The loopback interface declares only an ifnet structure, since it doesn't require any device-dependent data. We show the Ethernet and SLIP driver's softc structures with the array index of 0 in [Figure 3.20](#) since both drivers support multiple interfaces. The maximum number of interfaces of any given type is limited by a configuration parameter when the kernel is built.

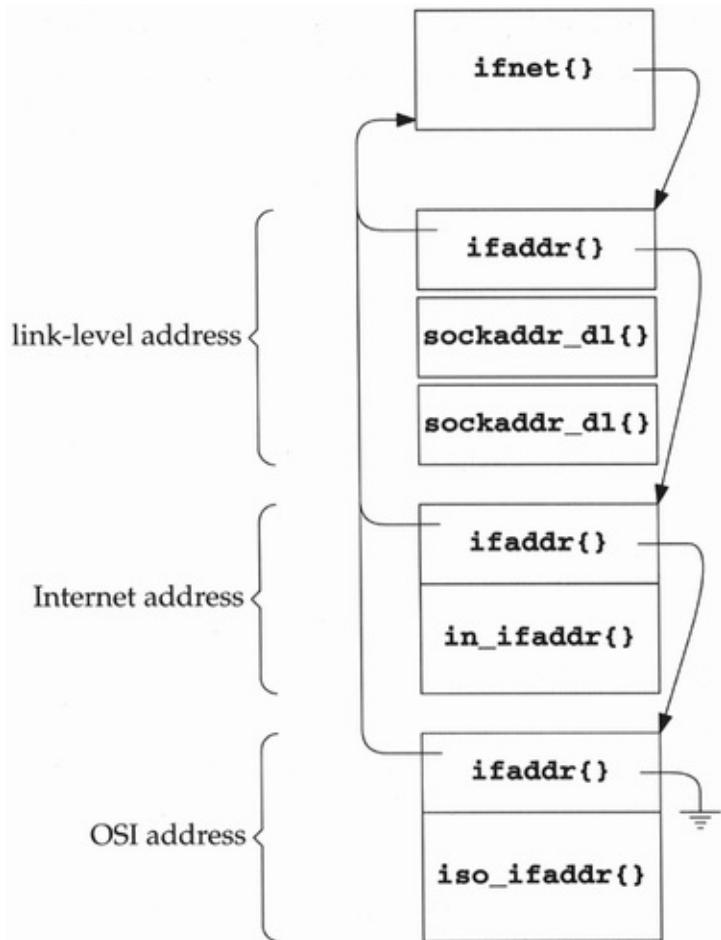
The arpcom structure ([Figure 3.26](#)) is common to all Ethernet drivers and contains information for the Address Resolution Protocol (ARP) and Ethernet multicasting. The le\_softc structure ([Figure 3.25](#)) contains additional information unique to the LANCE Ethernet device driver.

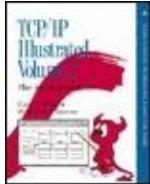
Each protocol stores addressing information for each interface in a list of specialized ifaddr structures. The Internet protocols use an in\_ifaddr structure ([Section 6.5](#)) and the OSI protocols an iso\_ifaddr structure. In addition to protocol addresses, the kernel assigns each interface a *link-level address* when the

interface is initialized, which identifies the interface within the kernel.

The kernel constructs the link-level address by allocating memory for an ifaddr structure and two sockaddr\_dl structuresone for the link-level address itself and one for the link-level address mask. The sockaddr\_dl structures are accessed by OSI, ARP, and the routing algorithms. [Figure 3.21](#) shows an Ethernet interface with a link-level address, an Internet address, and an OSI address. The construction and initialization of the link-level address (the ifaddr and the two sockaddr\_dl structures) is described in [Section 3.11](#).

**Figure 3.21. An interface address list containing link-level, Internet, and OSI addresses.**





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.7 Network Initialization Overview

All the structures we have described are allocated and attached to each other during kernel initialization. In this section we give a broad overview of the initialization steps. In later sections we describe the specific device- and protocol-initialization steps.

Some devices, such as the SLIP and loopback interfaces, are implemented entirely in software. These *pseudo-devices* are represented by a pdevinit structure ([Figure 3.22](#)) stored in the global pdevinit array. The array is constructed during kernel configuration. For example:

## Figure 3.22. pdevinit structure.

```
struct pdevinit pdevinit[] = {
    { slattach, 1 },
    { loopattach, 1 },
    { 0, 0 }
};

120 struct pdevinit {
121     void    (*pdev_attach) (int); /* attach function */
122     int     pdev_count;        /* number of devices */
123 };

```

*device.h*

*device.h*

120-123

In the pdevinit structures for the SLIP and the loopback interface, pdev\_attach is set to slattach and loopattach respectively. When the attach function is called, pdev\_count is passed as the only argument and specifies the number of devices to create. Only one loopback device is created but multiple SLIP devices may be created if the administrator configures the SLIP entry accordingly.

The network initialization functions from main are shown in [Figure 3.23](#).

## Figure 3.23. main function: network initialization.

---

```

70 main(framep)                                init_main.c
71 void    *framep;
72 {

    /* nonnetwork code */

96     cpu_startup();                      /* locate and initialize devices */

    /* nonnetwork code */

172     /* Attach pseudo-devices. (e.g., SLIP and loopback interfaces) */
173     for (pdev = pdevinit; pdev->pdev_attach != NULL; pdev++)
174         (*pdev->pdev_attach) (pdev->pdev_count);

175     /*
176      * Initialize protocols. Block reception of incoming packets
177      * until everything is ready.
178      */
179     s = splimp();
180     ifinit();                           /* initialize network interfaces */
181     domaininit();                     /* initialize protocol domains */
182     splx(s);

    /* nonnetwork code */

231     /* The scheduler is an infinite loop. */
232     scheduler();
233     /* NOTREACHED */
234 }

```

---

## 70-96

**cpu\_startup** locates and initializes all the hardware devices connected to the system, including any network interfaces.

## 97-174

After the kernel initializes the hardware devices, it calls each of the **pdev\_attach** functions contained within the **pdevinit** array.

## 175-234

ifinit and domaininit finish the initialization of the network interfaces and protocols and scheduler begins the kernel process scheduler, ifinit and domaininit are described in Chapter 7.

In the following sections we describe the initialization of the Ethernet, SLIP, and loopback interfaces.

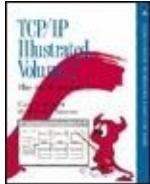
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.8 Ethernet Initialization

As part of `cpu_startup`, the kernel locates any attached network devices. The details of this process are beyond the scope of this text. Once a device is identified, a device-specific initialization function is called. [Figure 3.24](#) shows the initialization functions for our three sample interfaces.

**Figure 3.24. Network interface initialization functions.**

Device	Initialization Function
LANCE Ethernet	<code>leattach</code>
SLIP	<code>slattach</code>
loopback	<code>loopattach</code>

Each device driver for a network interface initializes a specialized ifnet structure and calls if\_attach to insert the structure into the linked list of interfaces. The le\_softc structure shown in [Figure 3.25](#) is the specialized ifnet structure for our sample Ethernet driver ([Figure 3.20](#)).

### **Figure 3.25. le\_softc structure.**

```
if_le.c
69 struct le_softc {
70     struct arpcom sc_ac;          /* common Ethernet structures */
71 #define sc_if    sc_ac.ac_if      /* network-visible interface */
72 #define sc_addr  sc_ac.ac_enaddr /* hardware Ethernet address */

    /* device-specific members */

95 } le_softc[NLE];
```

if\_le.c

## **le\_softc structure**

69-95

An array of le\_softc structures (with NLE elements) is declared in if\_le.c. Each structure starts with sc\_ac, an arpcom structure common to all Ethernet interfaces, followed by device-specific members. The sc\_if and sc\_addr macros

simplify access to the ifnet structure and Ethernet address within the arpcom structure, sc\_ac, shown in [Figure 3.26](#).

## Figure 3.26. arpcom structure.

```
————— if_ether.h
95 struct arpcom {
96     struct ifnet ac_if;          /* network-visible interface */
97     u_char   ac_enaddr[6];      /* ethernet hardware address */
98     struct in_addr ac_ipaddr;   /* copy of ip address - XXX */
99     struct ether_multi *ac_multiaddrs; /* list of ether multicast addrs */
100    int      ac_multicnt;       /* length of ac_multiaddrs list */
101 };
————— if_ether.h
```

## arpcom structure

95-101

The first member of the arpcom structure, ac\_if, is an ifnet structure as shown in [Figure 3.20](#). ac\_enaddr is the Ethernet hardware address copied by the LANCE device driver from the hardware when the kernel locates the device during cpu\_startup. For our sample driver, this occurs in the leattach function ([Figure 3.27](#)). ac\_ipaddr is the *last* IP address assigned to the device. We discuss address assignment in [Section 6.6](#), where we'll see

that an interface can have several IP addresses. See also [Exercise 6.3](#).  
ac\_multiaddrs is a list of Ethernet multicast addresses represented by ether\_multi structures. ac\_multicnt counts the entries in the list. The multicast list is discussed in [Chapter 12](#).

### Figure 3.27. leattach function.

```

106 leattach(hd)
107 struct hp_device *hd;
108 {
109     struct lereg0 *ler0;
110     struct lereg2 *ler2;
111     struct lereg2 *lemem = 0;
112     struct le_softc *le = &le_softc[hd->hp_unit];
113     struct ifnet *ifp = &le->sc_if;
114     char *cp;
115     int i;

116     /* device-specific code */

126     /*
127      * Read the ethernet address off the board, one nibble at a time.
128      */
129     cp = (char *) (lestd[3] + (int) hd->hp_addr);
130     for (i = 0; i < sizeof(le->sc_addr); i++) {
131         le->sc_addr[i] = (*++cp & 0xF) << 4;
132         cp++;
133         le->sc_addr[i] |= *++cp & 0xF;
134         cp++;
135     }
136     printf("le%d: hardware address %s\n", hd->hp_unit,
137           ether_sprintf(le->sc_addr));

138     /* device-specific code */

150     ifp->if_unit = hd->hp_unit;
151     ifp->if_name = "le";
152     ifp->if_mtu = ETHERMTU;
153     ifp->if_init = leinit;
154     ifp->if_reset = lereset;
155     ifp->if_ioctl = leioctl;
156     ifp->if_output = ether_output;
157     ifp->if_start = lestart;
158     ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
159     bpfattach(&ifp->if_bpf, ifp, DLT_EN10MB, sizeof(struct ether_header));
160     if_attach(ifp);
161     return (1);
162 }

```

if\_le.c

**106-115**

**Figure 3.27** shows the initialization code for the LANCE Ethernet driver.

The kernel calls `leattach` once for each LANCE card it finds in the system.

The single argument points to an

`hp_device` structure, which contains HP-specific information since this driver is written for an HP workstation.

`le` points to the specialized `ifnet` structure for the card ([Figure 3.20](#)) and `ifp` points to the first member of that structure, `sc_if`, a generic `ifnet` structure. The device-specific initializations are not included in [Figure 3.27](#) and are not discussed in this text.

## **Copy the hardware address from the device**

**126-137**

For the LANCE device, the Ethernet address assigned by the manufacturer is copied from the device to `sc_addr` (which is `sc_ac.ac_enaddr` see [Figure 3.26](#)) one nibble (4 bits) at a time in this for loop.

`lested` is a device-specific table of offsets to locate information relative to `hp_addr`, which points to LANCE-specific information.

The complete address is output to the

console by the printf statement to indicate that the device exists and is operational.

## Initialize the ifnet structure

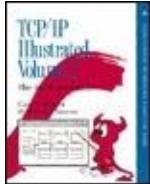
150-157

leattach copies the device unit number from the hp\_device structure into if\_unit to identify multiple interfaces of the same type. if\_name is "le" for this device; if\_mtu is 1500 bytes (ETHERMTU), the maximum transmission unit for Ethernet; if\_init, if\_reset, if\_ioctl, if\_output, and if\_start all point to device-specific implementations of the generic functions that control the network interface. [Section 4.1](#) describes these functions.

158

All Ethernet devices support IFF\_BROADCAST. The LANCE device does not receive its own transmissions, so IFF\_SIMPLEX is set. The driver and hardware supports multicasting so IFF\_MULTICAST is also set.

bpfattach registers the interface with BPF and is described with [Figure 31.8](#). The if\_attach function inserts the initialized ifnet structure into the linked list of interfaces ([Section 3.11](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.9 SLIP Initialization

The SLIP interface relies on a standard asynchronous serial device initialized within the call to `cpu_startup`. The SLIP pseudo-device is initialized when `main` calls `slattach` indirectly through the `pdev_attach` pointer in SLIP's `pdevinit` structure.

Each SLIP interface is described by an `sl_softc` structure shown in [Figure 3.28](#).

**Figure 3.28. `sl_softc` structure.**

```
43 struct sl_softc {
44     struct ifnet sc_if;          /* network-visible interface */
45     struct ifqueue sc_fastq;    /* interactive output queue */
46     struct tty *sc_ttyp;        /* pointer to tty structure */
47     u_char *sc_mp;             /* pointer to next available buf char */
48     u_char *sc_ep;             /* pointer to last available buf char */
49     u_char *sc_buf;            /* input buffer */
50     u_int sc_flags;            /* Figure 3.29 */
51     u_int sc_escape;           /* =1 if last char input was FRAME_ESCAPE */
52     struct slcompress sc_comp; /* tcp compression data */
53     caddr_t sc_bpf;            /* BPF data */
54 };
```

if\_slvar.h

## 43-54

As with all interface structures, sl\_softc starts with an ifnet structure followed by device-specific information.

In addition to the output queue found in the ifnet structure, a SLIP device maintains a separate queue, sc\_fastq, for packets requesting low-delay service typically generated by interactive applications.

sc\_ttyp points to the associated terminal device. The two pointers sc\_buf and sc\_ep point to the first and last bytes of the buffer for an incoming SLIP packet. sc\_mp points to the location for the next incoming byte and is advanced as additional bytes arrive.

The four flags defined by the SLIP driver

are shown in [Figure 3.29](#).

### Figure 3.29. SLIP if\_flags and sc\_flags values.

Constant	sc_softc member	Description
SC_COMPRESS	sc_if.if_flags	IFF_LINK0; compress TCP traffic
SC_NOICMP	sc_if.if_flags	IFF_LINK1; suppress ICMP traffic
SC_AUTOCOMP	sc_if.if_flags	IFF_LINK2; auto-enable TCP compression
SC_ERROR	sc_flags	error detected; discard incoming frame

SLIP defines the three interface flags reserved for the device driver in the ifnet structure and one additional flag defined in the sl\_softc structure.

sc\_escape is used by the IP encapsulation mechanism for serial lines ([Section 5.3](#)), while TCP header compression ([Section 29.13](#)) information is kept in sc\_comp.

The BPF information for the SLIP device is pointed to by sc\_bpf.

The sl\_softc structure is initialized by slattach, shown in [Figure 3.30](#).

### Figure 3.30. slattach function.

```
135 void  
136 slattach()  
137 {  
138     struct sl_softc *sc;  
139     int i = 0;  
140     for (sc = sl_softc; i < NSL; sc++) {  
141         sc->sc_if.if_name = "sl";  
142         sc->sc_if.if_next = NULL;  
143         sc->sc_if.if_unit = i++;  
144         sc->sc_if.if_mtu = SLMTU;  
145         sc->sc_if.if_flags =  
146             IFF_POINTOPOINT | SC_AUTOCOMP | IFF_MULTICAST;  
147         sc->sc_if.if_type = IFT_SLIP;  
148         sc->sc_if.if_ioctl = slioctl;  
149         sc->sc_if.if_output = sloutput;  
150         sc->sc_if.if_snd.ifq_maxlen = 50;  
151         sc->sc_fastq.ifq_maxlen = 32;  
152         if_attach(&sc->sc_if);  
153         bpfattach(&sc->sc_bpf, &sc->sc_if, DLT_SLIP, SLIP_HDRLEN);  
154     }  
155 }
```

if\_sl.c

## 135-152

Unlike leattach, which initializes only one interface at a time, the kernel calls slattach once and slattach initializes all the SLIP interfaces. Hardware devices are initialized as they are discovered by the kernel during cpu\_startup, while pseudo-devices are initialized all at once when main calls the pdev\_attach function for the device. if\_mtu for a SLIP device is 296 bytes (SLMTU). This accommodates the standard 20-byte IP header, the standard 20-byte TCP header, and 256 bytes of user data ([Section 5.3](#)).

A SLIP network consists of two interfaces at each end of a serial communication line.

slattach turns on IFF\_POINTOPOINT, SC\_AUTOCOMP, and IFF\_MULTICAST in if\_flags.

The SLIP interface limits the length of its output packet queue, if\_snd, to 50 and its own internal queue, sc\_fastq, to 32. [Figure 3.42](#) shows that the length of the if\_snd queue defaults to 50 (ifqmaxlen) if the driver does not select a length, so the initialization here is redundant.

The Ethernet driver doesn't set its output queue length explicitly and relies on ifinit ([Figure 3.42](#)) to set it to the system default.

if\_attach expects a pointer to an ifnet structure so slattach passes the address of sc\_if, an ifnet structure and the first member of the sl\_softc structure.

A special program, slattach, is run (from the /etc/netstart initialization file) after the kernel has been initialized and joins the SLIP interface and an asynchronous serial device by opening the serial device and issuing ioctl commands ([Section 5.3](#)).

**153-155**

For each SLIP device, slattach calls bpfattach to register the interface with BPF.

---

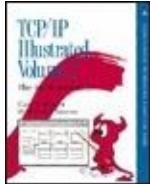
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.10 Loopback Initialization

Finally, we show the initialization for the single loopback interface. The loopback interface places any outgoing packets back on an appropriate input queue. There is no hardware device associated with the interface. The loopback pseudo-device is initialized when main calls loopattach indirectly through the pdev\_attach pointer in the loopback's pdevinit structure. [Figure 3.31](#) shows the loopattach function.

**Figure 3.31. Loopback interface initialization.**

```
41 void  
42 loopattach(n)  
43 int      n;  
44 {  
45     struct ifnet *ifp = &loif;  
46     ifp->if_name = "lo";  
47     ifp->if_mtu = LOMTU;  
48     ifp->if_flags = IFF_LOOPBACK | IFF_MULTICAST;  
49     ifp->if_ioctl = loioctl;  
50     ifp->if_output = looutput;  
51     ifp->if_type = IFT_LOOP;  
52     ifp->if_hdrlen = 0;  
53     ifp->if_addrlen = 0;  
54     if_attach(ifp);  
55     bpfattach(&ifp->if_bpf, ifp, DLT_NULL, sizeof(u_int));  
56 }
```

if\_loop.c

## 41-56

The loopback if\_mtu is set to 1536 bytes (LOMTU). In if\_flags, IFF\_LOOPBACK and IFF\_MULTICAST are set. A loopback interface has no link header or hardware address, so if\_hdrlen and if\_addrlen are set to 0. if\_attach finishes the initialization of the ifnet structure and bpfattach registers the loopback interface with BPF.

The loopback MTU should be at least 1576 ( $40 + 3 \times 512$ ) to leave room for a standard TCP/IP header. Solaris 2.3, for example, sets the loopback MTU to 8232 ( $40 + 8 \times 1024$ ). These calculations are biased toward the Internet protocols; other protocols may have default headers larger than 40 bytes.

---

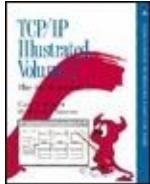
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



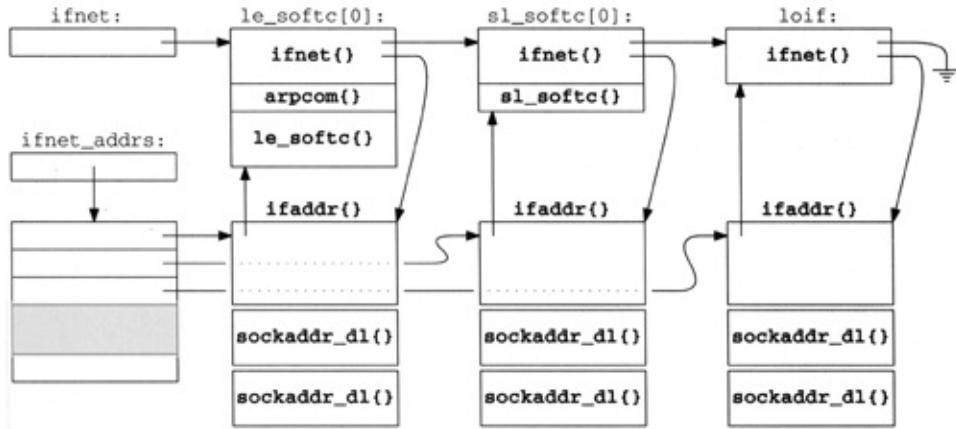
[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.11 if\_attach Function

The three interface initialization functions shown earlier each call `if_attach` to complete initialization of the interface's `ifnet` structure and to insert the structure on the list of previously configured interfaces. Also, in `if_attach`, the kernel initializes and assigns each interface a link-level address. [Figure 3.32](#) illustrates the data structures constructed by `if_attach`.

**Figure 3.32. ifnet list.**



In [Figure 3.32](#), `if_attach` has been called three times: from `leattach` with an `le_softc` structure, from `slattach` with an `sl_softc` structure, and from `loopattach` with a generic `ifnet` structure. Each time it is called it adds another `ifnet` structure to the `ifnet` list, creates a link-level `ifaddr` structure for the interface (which contains two `sockaddr_dl` structures, [Figure 3.33](#)), and initializes an entry in the `ifnet_addrs` array.

**Figure 3.33. `sockaddr_dl` structure.**

```

55 struct sockaddr_dl {
56     u_char    sdl_len;           /* Total length of sockaddr */
57     u_char    sdl_family;       /* AF_LINK */
58     u_short   sdl_index;        /* if != 0, system given index for
59                                interface */
60     u_char    sdl_type;         /* interface type (Figure 3.9) */
61     u_char    sdl_nlen;         /* interface name length, no trailing 0
62                                reqd. */
63     u_char    sdl_alen;         /* link level address length */
64     u_char    sdl_slen;         /* link layer selector length */
65     char      sdl_data[12];     /* minimum work area, can be larger;
66                                contains both if name and ll address */
67 };
68 #define LLADDR(s) ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))

```

if\_dl.h

The structures contained within le\_softc[0] and sl\_softc[0] are nested as shown in [Figure 3.20](#).

After this initialization, the interfaces are configured only with link-level addresses. IP addresses, for example, are not configured until much later by the ifconfig program ([Section 6.6](#)).

The link-level address contains a logical address for the interface and a hardware address if supported by the network (e.g., a 48-bit Ethernet address for le0). The hardware address is used by ARP and the OSI protocols, while the logical address within a sockaddr\_dl contains a name and numeric index for the interface within the kernel, which supports a table lookup for converting between an interface index and the associated ifaddr structure.

(ifa\_ifwithnet, [Figure 6.32](#)).

The sockaddr\_dl structure is shown in [Figure 3.33](#).

55-57

Recall from [Figure 3.18](#) that sdl\_len specifies the length of the entire address and sdl\_family specifies the address family, in this case AF\_LINK.

58

sdl\_index identifies the interface within the kernel. In [Figure 3.32](#) the Ethernet interface would have an index of 1, the SLIP interface an index of 2, and the loopback interface an index of 3. The global integer if\_index contains the last index assigned by the kernel.

60

sdl\_type is initialized from the if\_type member of the ifnet structure associated with this datalink address.

61-68

In addition to a numeric index, each interface has a text name formed from the `if_name` and `if_unit` members of the `ifnet` structure. For example, the first SLIP interface is called "sl0" and the second is called "sl1". The text name is stored at the front of the `sdl_data` array, and `sdl_nlen` is the length of this name in bytes (3 in our SLIP example).

The datalink address is also stored in the structure. The macro `LLADDR` converts a pointer to a `sockaddr_dl` structure into a pointer to the first byte beyond the text name. `sdl_alen` is the length of the hardware address. For an Ethernet device, the 48-bit hardware address appears in the `sockaddr_dl` structure beyond the text name. [Figure 3.38](#) shows an initialized `sockaddr_dl` structure.

Net/3 does not use `sdl_slen`.

`if_attach` updates two global variables. The first, `if_index`, holds the index of the last interface in the system and the second, `ifnet_addrs`, points to an array of `ifaddr` pointers. Each entry in the array points to

the link-level address of an interface. The array provides quick access to the link-level address for every interface in the system.

The `if_attach` function is long and consists of several tricky assignment statements. We describe it in four parts, starting with Figure 3.34.

### Figure 3.34. `if_attach` function: assign interface index.

```
59 void
60 if_attach(ifp)
61 struct ifnet *ifp;
62 {
63     unsigned socksize, ifasize;
64     int      namelen, unitlen, masklen, ether_output();
65     char    workbuf[12], *unitname;
66     struct ifnet **p = &ifnet; /* head of interface list */
67     struct sockaddr_dl *sdl;
68     struct ifaddr *ifa;
69     static int if_indexlim = 8; /* size of ifnet_addrs array */
70     extern void link_xtrequest();

71     while (*p)           /* find end of interface list */
72         p = &((*p)->if_next);
73     *p = ifp;
74     ifp->if_index = ++if_index; /* assign next index */

75     /* resize ifnet_addrs array if necessary */
76     if (ifnet_addrs == 0 || if_index >= if_indexlim) {
77         unsigned n = (if_indexlim <= 1) * sizeof(ifa);
78         struct ifaddr **q = (struct ifaddr **)
79             malloc(n, M_IFADDR, M_WAITOK);

80         if (ifnet_addrs) {
81             bcopy((caddr_t) ifnet_addrs, (caddr_t) q, n / 2);
82             free((caddr_t) ifnet_addrs, M_IFADDR);
83         }
84         ifnet_addrs = q;
85     }
```

59-74

if\_attach has a single argument, ifp, a pointer to the ifnet structure that has been initialized by a network device driver. Net/3 keeps all the ifnet structures on a linked list headed by the global pointer ifnet. The while loop locates the end of the list and saves the address of the null pointer at the end of the list in p. After the loop, the new ifnet structure is attached to the end of the ifnet list, if\_index is incremented, and the new index is assigned to ifp->if\_index.

## **Resize ifnet\_addrs array if necessary**

75-85

The first time through if\_attach, the ifnet\_addrs array doesn't exist so space for 16 entries ( $16 = 8 << 1$ ) is allocated. When the array becomes full, a new array of twice the size is allocated and the entries from the old array are copied to the new array.

`if_indexlim` is a static variable private to `if_attach`. `if_indexlim` is updated by the `<=>` operator.

The `malloc` and `free` functions in [Figure 3.34](#) are *not* the standard C library functions of the same name. The second argument in the kernel versions specifies a type, which is used by optional diagnostic code in the kernel to detect programming errors. If the third argument to `malloc` is `M_WAITOK`, the function blocks the calling process if it needs to wait for free memory to become available. If the third argument is `M_DONTWAIT`, the function does not block and returns a null pointer when no memory is available.

The next section of `if_attach`, shown in [Figure 3.35](#), prepares a text name for the interface and computes the size of the link-level address.

### **Figure 3.35. `if_attach` function: compute size of link-level address.**

```

86  /* create a Link Level name for this device */
87  unitname = sprint_d((u_int) ifp->if_unit, workbuf, sizeof(workbuf));
88  namelen = strlen(ifp->if_name);
89  unitlen = strlen(unitname);

90  /* compute size of sockaddr_dl structure for this device */
91 #define _offsetof(t, m) ((int)((caddr_t)&((t *)0)->m))
92  masklen = _offsetof(struct sockaddr_dl, sdl_data[0]) +
93      unitlen + namelen;
94  socksiz = masklen + ifp->if_addrhlen;
95 #define ROUNDUP(a) (1 + (((a) - 1) | (sizeof(long) - 1)))
96  socksiz = ROUNDUP(socksiz);
97  if (socksiz < sizeof(*sdl))
98      socksiz = sizeof(*sdl);
99  ifasize = sizeof(*ifa) + 2 * socksiz;

```

if.c

## Create link-level name and compute size of link-level address

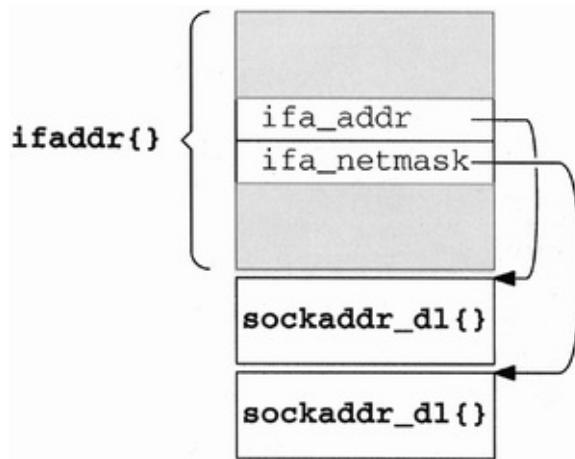
86-99

if\_attach constructs the name of the interface from if\_unit and if\_name. The function sprint\_d converts the numeric value of if\_unit to a string stored in workbuf. masklen is the number of bytes occupied by the information before sdl\_data in the sockaddr\_dl array plus the size of the text name for the interface (namelen + unitlen). The function rounds socksiz, which is masklen plus the hardware address length (if\_addrhlen), up to the boundary of a long integer (ROUNDUP). If this is less than the size of a sockaddr\_dl structure, the standard sockaddr\_dl structure is used, ifasize is the

size of an ifaddr structure plus two times socksiz, so it can hold the sockaddr\_dl structures.

In the next section, if\_attach allocates and links the structures together, as shown in Figure 3.36.

**Figure 3.36. The link-level address and mask assigned during if\_attach.**



In Figure 3.36 there is a gap between the ifaddr structure and the two sockaddr\_dl structures to illustrate that they are allocated in a contiguous area of memory but that they are not defined by a single C structure.

The organization shown in [Figure 3.36](#) is repeated in the `in_ifaddr` structure; the pointers in the generic `ifaddr` portion of the structure point to specialized `sockaddr` structures allocated in the device-specific portion of the structure, in this case, `sockaddr_dl` structures. [Figure 3.37](#) shows the initialization of these structures.

**Figure 3.37. `if_attach` function: allocate and initialize link-level address.**

```
100  if (ifa = (struct ifaddr *) malloc(ifasize, M_IFADDR, M_WAITOK)) {                                if.c
101      bzero((caddr_t) ifa, ifasize);
102      /* First: initialize the sockaddr_dl address */
103      sdl = (struct sockaddr_dl *) (ifa + 1);
104      sdl->sdl_len = socksize;
105      sdl->sdl_family = AF_LINK;
106      bcopy(ifp->if_name, sdl->sdl_data, namelen);
107      bcopy(unitname, namelen + (caddr_t) sdl->sdl_data, unitlen);
108      sdl->sdl_nlen = (namelen += unitlen);
109      sdl->sdl_index = ifp->if_index;
110      sdl->sdl_type = ifp->if_type;
111      ifnet_addrs[if_index - 1] = ifa;
112      ifa->ifa_ifp = ifp;
113      ifa->ifa_next = ifp->if_addrlist;
114      ifa->ifa_rtrequest = link_rtrequest;
115      ifp->if_addrlist = ifa;
116      ifa->ifa_addr = (struct sockaddr *) sdl;
117      /* Second: initialize the sockaddr_dl mask */
118      sdl = (struct sockaddr_dl *) (socksize + (caddr_t) sdl);
119      ifa->ifa_netmask = (struct sockaddr *) sdl;
120      sdl->sdl_len = masklen;
121      while (namelen != 0)
122          sdl->sdl_data[--namelen] = 0xff;
123  }
```

## The address

100-116

If enough memory is available, bzero fills the new structure with 0s and sdl points to the first sockaddr\_dl just after the ifaddr structure. If no memory is available, the code is skipped.

sdl\_len is set to the length of the sockaddr\_dl structure, and sdl\_family is set to AF\_LINK. A text name is constructed within sdl\_data from if\_name and unitname, and the length is saved in sdl\_nlen. The interface's index is copied into sdl\_index as well as the interface type into sdl\_type. The allocated structure is inserted into the ifnet\_addrs array and linked to the ifnet structure by ifa\_ifp and if\_addrlist. Finally, the sockaddr\_dl structure is connected to the ifnet structure with ifa\_addr. Ethernet interfaces replace the default function, link\_rtrequest with arp\_rtrequest. The loopback interface installs loop\_rtrequest. We describe ifa\_rtrequest and arp\_rtrequest in [Chapters 19 and 21](#). link\_rtrequest and loop\_rtrequest are left for readers to investigate on their own. This completes

the initialization of the first sockaddr\_dl structure.

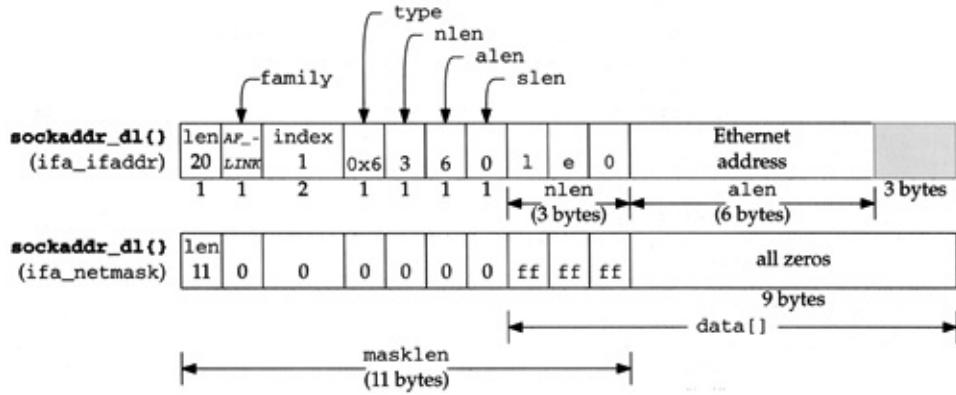
## The mask

117-123

The second sockaddr\_dl structure is a bit mask that selects the text name that appears in the first structure. ifa\_netmask from the ifaddr structure points to the mask structure (which in this case selects the interface text name and not a network mask). The while loop turns on the bits in the bytes corresponding to the name.

[Figure 3.38](#) shows the two initialized sockaddr\_dl structures for our example Ethernet interface, where if\_name is "le", if\_unit is 0, and if\_index is 1.

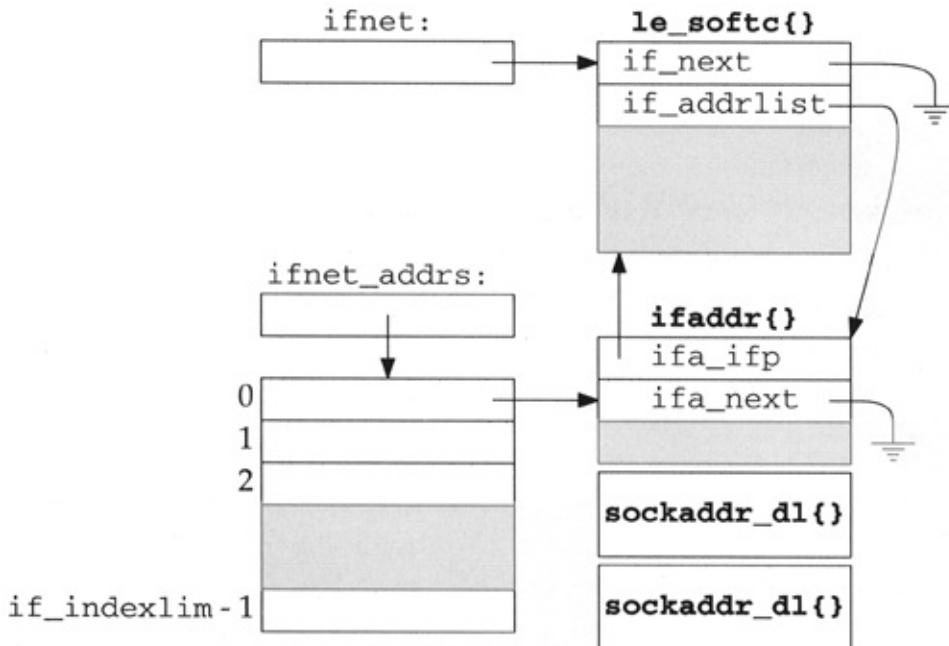
**Figure 3.38. The initialized Ethernet sockaddr\_dl structures (sdl\_prefix omitted).**



In [Figure 3.38](#), the address is shown after `ether_ifattach` has done additional initialization of the structure ([Figure 3.41](#)).

[Figure 3.39](#) shows the structures after the first interface has been attached by `if_attach`.

**Figure 3.39. The ifaddr and sockaddr\_dl structures after if\_attach is called for the first time.**



At the end of `if_attach`, the `ether_ifattach` function is called for Ethernet devices, as shown in [Figure 3.40](#).

**Figure 3.40. `if_attach` function: Ethernet initialization.**

```

124     /* XXX -- Temporary fix before changing 10 ethernet drivers */
125     if (ifp->if_output == ether_output)
126         ether_ifattach(ifp);
127 }

```

124-127

`ether_ifattach` isn't called earlier (from `leattach`, for example) because it copies the Ethernet hardware address into the

sockaddr\_dl allocated by if\_attach.

The XXX comment indicates that the author found it easier to insert the code here once than to modify all the Ethernet drivers.

## ether\_ifattach function

The ether\_ifattach function performs the ifnet structure initialization common to all Ethernet devices.

**Figure 3.41. ether\_ifattach function.**

```
if_ETHERSUBR.C
338 void
339 ether_ifattach(ifp)
340 struct ifnet *ifp;
341 {
342     struct ifaddr *ifa;
343     struct sockaddr_dl *sdl;
344     ifp->if_type = IFT_ETHER;
345     ifp->if_addrlen = 6;
346     ifp->if_hdrlen = 14;
347     ifp->if_mtu = ETHERMTU;
348     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
349         if ((sdl = (struct sockaddr_dl *) ifa->ifa_addr) &&
350             sdl->sdl_family == AF_LINK) {
351             sdl->sdl_type = IFT_ETHER;
352             sdl->sdl_alen = ifp->if_addrlen;
353             bcopy((caddr_t) ((struct arpcom *) ifp)->ac_enaddr,
354                   LLADDR(sdl), ifp->if_addrlen);
355             break;
356         }
357 }
```

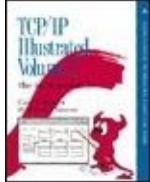
338-357

For an Ethernet device, if\_type is IFT\_ETHER, the hardware address is 6 bytes long, the entire Ethernet header is 14 bytes in length, and the Ethernet MTU is 1500 (ETHERMTU).

The MTU was already assigned by leattach, but other Ethernet device drivers may not have performed this initialization.

Section 4.3 discusses the Ethernet frame organization in more detail. The for loop locates the link-level address for the interface and then initializes the Ethernet hardware address information in the sockaddr\_dl structure. The Ethernet address that was copied into the arpcom structure during system initialization is now copied into the link-level address.





TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.12 ifinit Function

After the interface structures are initialized and linked together, main (Figure 3.23) calls ifinit, shown in Figure 3.42.

Figure 3.42. ifinit function.

```
if.c
43 void
44 ifinit()
45 {
46     struct ifnet *ifp;
47     for (ifp = ifnet; ifp; ifp = ifp->if_next)
48         if (ifp->if_snd.ifq_maxlen == 0)
49             ifp->if_snd.ifq_maxlen = ifqmaxlen; /* set default length */
50     if_slowtimo(0);
51 }
```

if.c

43-51

The for loop traverses the interface list and sets the maximum size of each interface

output queue to 50 (ifqmaxlen) if it hasn't already been set by the interface's attach function.

An important consideration for the size of the output queue is the number of packets required to send a maximum-sized datagram. For Ethernet, if a process calls `sendto` with 65,507 bytes of data, it is fragmented into 45 fragments and each fragment is put onto the interface output queue. If the queue were much smaller, the process could never send that large a datagram, as the queue wouldn't have room.

`if_slowtimo` starts the interface watchdog timers. When an interface timer expires, the kernel calls the watchdog function for the interface. An interface can reset the timer periodically to prevent the watchdog function from being called, or set `if_timer` to 0 if the watchdog function is not needed. [Figure 3.43](#) shows the `if_slowtimo` function.

### Figure 3.43. `if_s1owtimo` function.

```
338 void
339 if_slowtimo(arg)
340 void    *arg;
341 {
342     struct ifnet *ifp;
343     int      s = splimp();
344
345     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
346         if (ifp->if_timer == 0 || --ifp->if_timer)
347             continue;
348         if (ifp->if_watchdog)
349             (*ifp->if_watchdog) (ifp->if_unit);
350     }
351     splx(s);
352     timeout(if_slowtimo, (void *) 0, hz / IFNET_SLOWHZ);
353 }
```

if.c

if.c

## 338-343

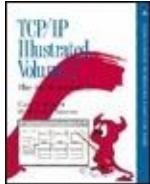
The single argument, arg, is not used but is required by the prototype for the slow timeout functions ([Section 7.4](#)).

## 344-352

if\_slowtimo ignores interfaces with if\_timer equal to 0; if if\_timer does not equal 0, if\_slowtimo decrements if\_timer and calls the if\_watchdog function associated with the interface when the timer reaches 0. Packet processing is blocked by splimp during if\_slowtimo. Before returning, ip\_slowtimo calls timeout to schedule a call to itself in hz/IFNET\_SLOWHZ clock ticks, hz is the number of clock ticks that occur in 1 second (often 100). It is set at system initialization and remains constant

thereafter. Since IFNET\_SLOWHZ is defined to be 1, the kernel calls if\_slowtimo once every hz clock ticks, which is once per second.

The functions scheduled by the timeout function are called back by the kernel's callout function. See [[Leffler et al. 1989](#)] for additional details.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 3. Interface Layer

### 3.13 Summary

In this chapter we have examined the ifnet and ifaddr structures that are allocated for each network interface found at system initialization time. The ifnet structures are linked into the ifnet list. The link-level address for each interface is initialized, attached to the ifnet structure's address list, and entered into the if\_addrs array.

We discussed the generic sockaddr structure and its sa\_family, and sa\_len members, which specify the type and length of every address. We also looked at the initialization of the sockaddr\_dl structure for a link-level address.

In this chapter, we introduced the three example network interfaces that we use throughout the book.

## Exercises

The netstat program on many Unix systems lists network interfaces and their configuration. Try netstat -i on a **3.1** system you have access to. What are the names (if\_name) and maximum transmission units (if\_mtu) of the network interfaces?

In if\_slowtimo ([Figure 3.43](#)) the splimp and splx calls appear outside the loop. What are the advantages and disadvantages of this arrangement compared with placing the calls within the loop? **3.2**

Why is SLIP's interactive queue **3.3** shorter than SLIP'S standard output queue?

Why aren't if\_hdrlen and if\_addralen  
**3.4** initialized in slattach?

**3.5** Draw a picture similar to [Figure 3.38](#) for the SLIP and loopback devices.

---

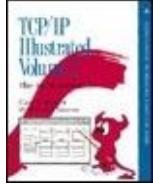
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 4. Interfaces: Ethernet

[Section 4.1. Introduction](#)

[Section 4.2. Code Introduction](#)

[Section 4.3. Ethernet Interface](#)

[Section 4.4. ioctl System Call](#)

[Section 4.5. Summary](#)

---

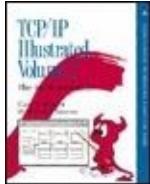
**Team-Fly**



[Previous](#)

[Next](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 4. Interfaces: Ethernet

### 4.1 Introduction

In [Chapter 3](#) we discussed the data structures used by all interfaces and the initialization of those data structures. In this chapter we show how the Ethernet device driver operates once it has been initialized and is receiving and transmitting frames. The second half of this chapter covers the generic ioctl commands for configuring network devices. [Chapter 5](#) covers the SLIP and loopback drivers.

We won't go through the entire source code for the Ethernet driver, since it is around 1,000 lines of C code (half of which is concerned with the hardware details of one particular interface card), but we do

look at the device-independent Ethernet code and how the driver interfaces with the rest of the kernel.

If the reader is interested in going through the source code for a driver, the Net/3 release contains the source code for many different interfaces. Access to the interface's technical specifications is required to understand the device-specific commands. [Figure 4.1](#) shows the various drivers provided with Net/3, including the LANCE driver, which we discuss in this text.

**Figure 4.1. Ethernet drivers available in Net/3.**

Device	File
DEC DEUNA Interface	vax/if/if_de.c
3Com Ethernet Interface	vax/if/if_ec.c
Excelan EXOS 204 Interface	vax/if/if_ex.c
Interlan Ethernet Communications Controller	vax/if/if_il.c
Interlan NP100 Ethernet Communications Controller	vax/if/if_ix.c
Digital Q-BUS to NI Adapter	vax/if/if_qe.c
CMC ENP-20 Ethernet Controller	tahoe/if/if_enp.c
Excelan EXOS 202(VME) & 203(QBUS)	tahoe/if/if_ex.c
ACC VERSAbus Ethernet Controller	tahoe/if/if_ace.c
<b>AMD 7990 LANCE Interface</b>	<b>hp300/dev/if_le.c</b>
NE2000 Ethernet	i386/isa/if_ne.c
Western Digital 8003 Ethernet Adapter	i386/isa/if_we.c

Network device drivers are accessed through the seven function pointers in the ifnet structure ([Figure 3.11](#)). [Figure 4.2](#) lists the entry points to our three example drivers.

## Figure 4.2. Interface functions for the example drivers.

ifnet	Ethernet	SLIP	Loopback	Description
if_init	leinit			hardware initialization
if_output	ether_output			accept and queue frame for transmission
if_start	lestart			begin transmission of frame
if_done				output complete (unused)
if_ioctl	leioctl	sliioctl	loioctl	handle ioctl commands from a process
if_reset				reset the device to a known state
if_watchdog	lereset			watch the device for failures or collect statistics

Input functions are not included in [Figure 4.2](#) as they are interrupt-driven for network devices. The configuration of interrupt service routines is hardware-dependent and beyond the scope of this book. We'll identify the functions that handle device interrupts, but not the mechanism by which these functions are invoked.

Only the if\_output and if\_ioctl functions are called with any consistency. if\_init,

`if_done`, and `if_reset` are never called or only called from device-specific code (e.g., `leinit` is called directly by `leioctl`). `if_start` is called only by the `ether_output` function.

---

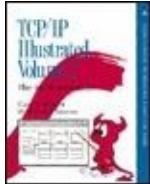
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 4. Interfaces: Ethernet

## 4.2 Code Introduction

The code for the Ethernet device driver and the generic interface ioctls resides in two headers and three C files, which are listed in [Figure 4.3](#).

**Figure 4.3. Files discussed in this chapter.**

File	Description
netinet/if_ether.h net/if.h	Ethernet structures ioctl command definitions
net/if_ETHERSUBR.C hp300/dev/if_le.c net/if.c	generic Ethernet functions LANC Ethernet driver ioctl processing

## Global Variables

The global variables shown in [Figure 4.4](#) include the protocol input queues, the LANCE interface structure, and the Ethernet broadcast address.

## Figure 4.4. Global variables introduced in this chapter.

Variable	Datatype	Description
arpintrq	struct ifqueue	ARP input queue
clnlintrq	struct ifqueue	CLNP input queue
ipintrq	struct ifqueue	IP input queue
le_softc	struct le_softc []	LANCE Ethernet interface
etherbroadcastaddr	u_char []	Ethernet broadcast address

le\_softc is an array, since there can be several Ethernet interfaces.

## Statistics

The statistics collected in the ifnet structure for each interface are described in [Figure 4.5](#).

## Figure 4.5. Statistics maintained in the ifnet structure.

ifnet member	Description	Used by SNMP
if_collisions	#collisions on CSMA interfaces	
if_ibytes	total #bytes received	•
if_ierrors	#packets received with input errors	•
if_imcasts	#packets received as multicasts or broadcasts	•
if_ipackets	#packets received on interface	•
if_iqdrops	#packets dropped on input, by this interface	•
if_lastchange	time of last change to statistics	•
if_noproto	#packets destined for unsupported protocol	•
if_obytes	total #bytes sent	•
if_oerrors	#output errors on interface	•
if_omcasts	#packets sent as multicasts	•
if_opackets	#packets sent on interface	•
if_snd.ifq_drops	#packets dropped during output	•
if_snd.ifq_len	#packets in output queue	•

**Figure 4.6** shows some sample output from the netstat command, which includes statistics from the ifnet structure.

## Figure 4.6. Sample interface statistics.

netstat -i output								
Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
le0	1500	<Link>	8.0.9.13.d.33	28680519	814	29234729	12	942798
le0	1500	128.32.33	128.32.33.5	28680519	814	29234729	12	942798
s10*	296	<Link>		54036	0	45402	0	0
s10*	296	128.32.33	128.32.33.5	54036	0	45402	0	0
s11	296	<Link>		40397	0	33544	0	0
s11	296	128.32.33	128.32.33.5	40397	0	33544	0	0
s12*	296	<Link>		0	0	0	0	0
s13*	296	<Link>		0	0	0	0	0
lo0	1536	<Link>		493599	0	493599	0	0
lo0	1536	127	127.0.0.1	493599	0	493599	0	0

The first column contains if\_name and if\_unit displayed as a string. If the interface is shut down (IFF\_UP is not set), an asterisk appears next to the name. In

## Figure 4.6, sl0, sl2, and sl3 are shut down.

The second column shows if\_mtu. The output under the "Network" and "Address" headings depends on the type of address. For link-level addresses, the contents of sdl\_data from the sockaddr\_dl structure are displayed. For IP addresses, the subnet and unicast addresses are displayed. The remaining columns are if\_ipackets, if\_ierrors, if\_opackets, if\_oerrors, and if\_collisions.

- Approximately 3% of the packets collide on output ( $942,798/29,234,729 = 3\%$ ).
- The SLIP output queues are never full on this machine since there are no output errors for the SLIP interfaces.
- The 12 Ethernet output errors are problems detected by the LANCE hardware during transmission. Some of these errors may also be counted as collisions.
- The 814 Ethernet input errors are also problems detected by the hardware,

such as packets that are too short or that have invalid checksums.

## SNMP Variables

Figure 4.7 shows a single interface entry object (ifEntry) from the SNMP interface table (ifTable), which is constructed from the ifnet structures for each interface.

**Figure 4.7. Variables in interface table:  
ifTable.**

Interface table, index = < ifIndex >		
SNMP variable	ifnet member	Description
ifIndex	if_index	uniquely identifies the interface
ifDescr	if_name	text name of interface
ifType	if_type	type of interface (e.g., Ethernet, SLIP, etc.)
ifMtu	if_mtu	MTU of the interface in bytes
ifSpeed	(see text)	nominal speed of the interface in bits per second
ifPhysAddress	ac_enaddr	media address (from arpcom structure)
ifAdminStatus	(see text)	desired state of the interface (IFF_UP flag)
ifOperStatus	if_flags	operational state of the interface (IFF_UP flag)
ifLastChange	(see text)	last time the statistics changed
ifInOctets	if_ibytes	total #input bytes
ifInUcastPkts	if_ipackets - if_imcasts	#input unicast packets
ifInNUcastPkts	if_imcasts	#input broadcast or multicast packets
ifInDiscards	if_iqdrops	#packets discarded because of implementation limits
ifInErrors	if_ierrors	#packets with errors
ifInUnknownProtos	if_noproto	#packets destined to an unknown protocol
ifOutOctets	if_obytes	#output bytes
ifOutUcastPkts	if_opackets - if_omcasts	#output unicast packets
ifOutNUcastPkts	if_omcasts	#output broadcast or multicast packets
ifOutDiscards	if_snd.ifq_drops	#output packets dropped because of implementation limits
ifOutErrors	if_oerrors	#output packets dropped because of errors
ifOutQLen	if_snd.ifq_len	output queue length
ifSpecific	n/a	SNMP object ID for media-specific information (not implemented)

The ISODE SNMP agent derives ifSpeed from if\_type and maintains an internal variable for ifAdminStatus. The agent reports ifLastChange based on if\_lastchange in the ifnet structure but relative to the agent's boot time, not the boot time of the system. The agent returns a null variable for ifSpecific.

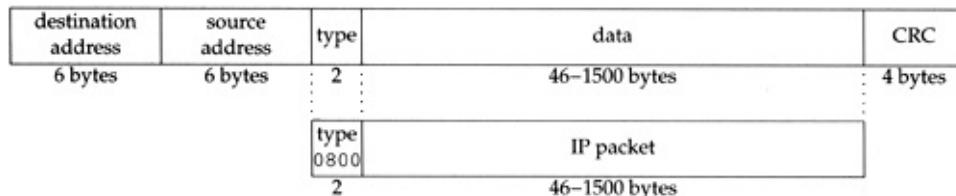
## Chapter 4. Interfaces: Ethernet

### 4.3 Ethernet Interface

Net/3 Ethernet device drivers all follow the same basic design. All Net/3 device drivers because the writer of a driver for one card can copy it and modify it. In this section, we will introduce the IEEE 802.3 Ethernet standard and outline the design of an Ethernet interface driver. Figure 4.8 illustrates the design.

Figure 4.8 illustrates Ethernet encapsulation of an IP packet.

**Figure 4.8. Ethernet encapsulation of an IP packet**



Ethernet frames consist of 48-bit destination and source addresses that identify the format of the data carried by the frame.

(2048). The frame is terminated with a 32-bit C in the frame.

We are describing the original Ethernet framir Equipment Corp., Intel Corp., and Xerox Corp TCP/IP networks. An alternative form is specif Electronics Engineers) 802.2 and 802.3 stand differences between the two forms. See [Stall standards.

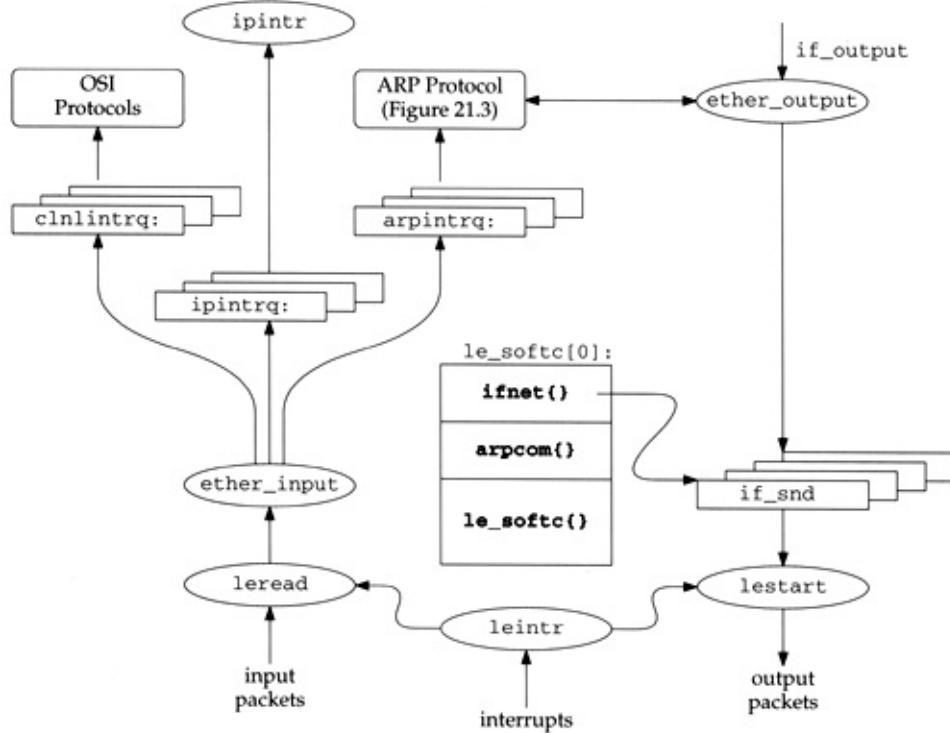
Encapsulation of IP packets for Ethernet is spe networks by RFC 1042 [Postel and Reynolds 1

We will refer to the 48-bit Ethernet addresses as hardware addresses is done by the ARP protocol [1982]) and from hardware to IP addresses by the ARP protocol [1984]). Ethernet addresses come in two types, single Ethernet interface, and a multicast address. An Ethernet *broadcast* is a multicast received by all devices on the network and assigned by the device's manufacturer, although it can be changed by software.

Some DECNET protocols require the hardware address to be changed, so DECNET must be able to change the Ethernet address.

Figure 4.9 illustrates the data structures and fu

**Figure 4.9. Eth**



In figures, a function is identified by an ellipse and a group of functions by a rounded box (A

In the top left corner of [Figure 4.9](#) we show the Layer (clnl) protocol, IP, and ARP. We won't say emphasize that ether\_input demultiplexes Ethe

Technically, OSI uses the term Connectionless show the terminology used by the Net/3 code [Stallings 1993] summarizes the standard.

The le\_softc interface structure is in the center and arpcom portions of the structure. The rema We showed the ifnet structure in [Figure 3.6](#) and

## leintr Function

We start with the reception of Ethernet frames. initialized and the system has been configured an interrupt. In normal operation, an Ethernet hardware address and for the Ethernet broadcast the interface generates an interrupt and the kernel

In [Chapter 12](#), we'll see that many Ethernet interfaces receive multicast frames (other than broadcasts).

Some interfaces can be configured to run in promiscuous mode, receiving all frames that appear on the network. The tc advantage of this feature using BPF.

leintr examines the hardware and, if a frame has arrived, copies it from the interface to a chain of mbufs (with m\_devgiven). If the transmission has completed or an error has been detected, the driver updates the appropriate interface statistics, releases the mbufs, and returns to transmit another frame.

All Ethernet device drivers deliver their received mbuf chain constructed by the device driver do as a separate argument to ether\_input. The eth

**Figure 4.10. The ether\_input function**

```
38 struct ether_header {  
39     u_char ether_dhost[6];      /* Ethernet destination address */  
40     u_char ether_shost[6];      /* Ethernet source address */  
41     u_short ether_type;        /* Ethernet frame type */  
42 };
```

*if\_ether.h*

```
if_ether.h
```

## 38-42

The Ethernet CRC is not generally available. It hardware, which discards frames that arrive with responsible for converting ether\_type between it is always in host byte order.

## Ieread Function

The leread function ([Figure 4.11](#)) starts with a and constructs an ether\_header structure and a the Ethernet frame. leread also passes the inco

**Figure 4.11.**

```

528 leread(unit, buf, len)                                if_le.c
529 int      unit;
530 char    *buf;
531 int      len;
532 {
533     struct le_softc *le = &le_softc[unit];
534     struct ether_header *et;
535     struct mbuf *m;
536     int      off, resid, flags;
537     le->sc_if.if_ipackets++;
538     et = (struct ether_header *) buf;
539     et->ether_type = ntohs((u_short) et->ether_type);
540     /* adjust input length to account for header and CRC */
541     len = len - sizeof(struct ether_header) - 4;
542     off = 0;
543     if (len <= 0) {
544         if (ledebug)
545             log(LOG_WARNING,
546                 "le%d: ierror(runt packet): from %s: len=%d\n",
547                 unit, ether_sprintf(et->ether_shost), len);
548         le->sc_runt++;
549         le->sc_if.if_ierrors++;
550         return;
551     }
552     flags = 0;
553     if (bcmcmp((caddr_t) etherbroadcastaddr,
554                 (caddr_t) et->ether_dhost, sizeof(etherbroadcastaddr)) == 0)
555         flags |= M_BCAST;
556     if (et->ether_dhost[0] & 1)
557         flags |= M_MCAST;
558     /*
559     * Check if there's a bpf filter listening on this interface.
560     * If so, hand off the raw packet to enet.
561     */
562     if (le->sc_if.if_bpf) {
563         bpf_tap(le->sc_if.if_bpf, buf, len + sizeof(struct ether_header));
564         /*
565          * Keep the packet if it's a broadcast or has our
566          * physical ethernet address (or if we support
567          * multicast and it's one).
568          */
569         if ((flags & (M_BCAST | M_MCAST)) == 0 &&
570             bcmcmp(et->ether_dhost, le->sc_addr,
571                   sizeof(et->ether_dhost)) != 0)
572             return;
573     }
574     /*
575      * Pull packet off interface. Off is nonzero if packet
576      * has trailing header; m_devget will then force this header
577      * information to be at the front, but we still have to drop
578      * the type and length which are at the front of any trailer data.
579      */
580     m = m_devget((char *) (et + 1), len, off, &le->sc_if, 0);
581     if (m == 0)
582         return;
583     m->m_flags |= flags;
584     ether_input(&le->sc_if, et, m);
585 }

```

The leintr function passes three arguments to lcbmp: dev, which is the card that received a frame; buf, which points to the data bytes in the frame (including the header and the CRC); and len, which is the length of the frame.

The function constructs the ether\_header structure by copying the first 14 bytes of buf into ether\_header, converting the Ethernet type value to host byte order, and then skipping the remaining bytes.

## 540-551

The number of data bytes is computed by subtracting the size of the header and the CRC from len. *Runt packets*, which are too short to contain a payload, are discarded.

## 552-557

Next, the destination address is examined to determine if it is an Ethernet multicast address. The Ethernet broadcast address is a multicast address; it has every bit set. etherbroadcastaddr is defined as:

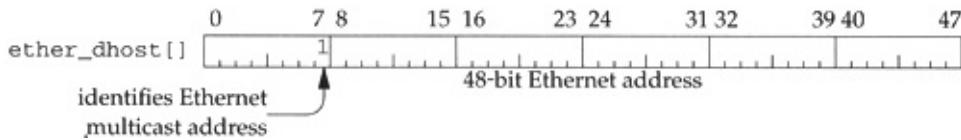
```
u_char etherbroadcastaddr[6] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF }
```

This is a convenient way to define a 48-bit value, since the individual characters are 8-bit values something that is not possible with a single integer variable.

If bcmp reports that etherbroadcastaddr and ether\_header.dhost are equal,

An Ethernet multicast addresses is identified by the address. [Figure 4.12](#) illustrates this.

## Figure 4.12. Testing for a



In [Chapter 12](#) we'll see that not all Ethernet multicasts must examine the packet further.

If the multicast bit is on in the address, `M_MCA` tests is important: first `ether_input` compares the address, and if they are different it checks the address against an Ethernet multicast address ([Exercise 4.1](#)).

### 558-573

If the interface is tapped by BPF, the frame is passed to `ether_input`. For SLIP and the loopback interfaces, a special case is made: they do not have a link-level header (unlike Ethernet).

When an interface is tapped by BPF, it can be configured to capture all Ethernet frames that appear on the network. If the hardware address in the frame does not match the interface's address, the packet is discarded.

### 574-585

`m_devget` ([Section 2.6](#)) copies the data from the buffer that `ether_input` allocates. The first argument to `m_devget` points to the start of the buffer.

is the first data byte in the frame. If `m_devget` is non-zero, the driver processes the packet.

## ether\_input Function

`ether_input`, shown in Figure 4.13, examines the received data and then queues the packet to the appropriate queue.

Figure 4.13. `ether_input` Function

```
196 void  
197 ether_input(ifp, eh, m)  
198 struct ifnet *ifp;  
199 struct ether_header *eh;  
200 struct mbuf *m;  
201 {  
202     struct ifqueue *inq;  
203     struct llc *l;  
204     struct arpcom *ac = (struct arpcom *) ifp;  
205     int     s;  
206     if ((ifp->if_flags & IFF_UP) == 0) {  
207         m_freem(m);  
208         return;  
209     }  
210     ifp->if_lastchange = time;
```

```

211     ifp->if_ibytes += m->m_pkthdr.len + sizeof(*eh);
212     if (bcmpl((caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
213                 sizeof(etherbroadcastaddr)) == 0)
214         m->m_flags |= M_BCAST;
215     else if (eh->ether_dhost[0] & 1)
216         m->m_flags |= M_MCAST;
217     if (m->m_flags & (M_BCAST | M_MCAST))
218         ifp->if_imcasts++;
219
220     switch (eh->ether_type) {
221     case ETHERTYPE_IP:
222         schednetisr(NETISR_IP);
223         inq = &ipintrq;
224         break;
225
226     case ETHERTYPE_ARP:
227         schednetisr(NETISR_ARP);
228         inq = &arpintrq;
229         break;
230
231     default:
232         if (eh->ether_type > ETHERMTU) {
233             m_freem(m);
234             return;
235         }
236
237     }
238
239     /* OSI code */
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308     s = splimp();
309     if (IF_QFULL(inq)) {
310         IF_DROP(inq);
311         m_freem(m);
312     } else
313         IF_ENQUEUE(inq, m);
314     splx(s);
315. }

```

*if\_ETHERSUBR.C*

## Broadcast and multicast recognition

196-209

The arguments to ether\_input are ifp, a pointer to the Ethernet header of the received packet (excluding the Ethernet header).

Any packets that arrive on an inoperative interface have been configured with a protocol address, obtained from the ifconfig(8) program ([Section 6.6](#)).

## 210-218

The variable time is a global timeval structure to date, as the number of seconds and microseconds since 1970, Coordinated Universal Time [UTC]). A brief history [Ramsey 1993]. We'll encounter the timeval structure:

```
struct timeval {  
    long tv_sec;          /* seconds */  
    long tv_usec;         /* and microseconds */  
};
```

ether\_input updates if\_lastchange with the current time when it receives an incoming packet (the packet length plus the 14 bytes of overhead).

Next, ether\_input repeats the tests done by ler\_rx() for each multicast packet.

Some kernels may not have been compiled with ether\_input.

## Link-level demultiplexing

## 219-227

ether\_input jumps according to the Ethernet type. It handles the IP software interrupt and the IP input queue, ip\_rcv().

software interrupt is scheduled and arpintrq is :

An *isr* is an interrupt service routine.

In previous BSD releases, ARP packets were received at the interrupt level by calling arpinput directly. By default, they are handled at the software interrupt level.

If other Ethernet types are to be handled, a kernel module can be loaded here. Alternately, a process can receive other types of packets. These two methods are normally implemented using BPF (Berkeley Packet Filter).

228-307

The default case processes unrecognized Ethernet frames according to the 802.3 standard (such as the OUI field and the 802.3 *length* field occupy the same bytes). Encapsulations can be distinguished because the ranges are distinct from the range of lengths in the 802.3 standard. [Stallings 1993] contains a descriptive table of these ranges.

**Figure 4.14. Ethernet type ranges**

Range	Description
0 — 1500	IEEE 802.3 <i>length</i> field
1501 — 65535	Ethernet <i>type</i> field: IP packet
2048	
2054	ARP packet

There are many additional Ethernet type values. We'll show them in [Figure 4.14](#). RFC 1700 [Reynolds et al.] lists the common types.

## Queue the packet

308-315

Finally, ether\_input places the packet on the selected queue if it is not full. We'll see in [Figures 7.23](#) and [21.16](#) that the queue can hold up to `(ipqmaxlen)` packets each.

When ether\_input returns, the device driver tells the kernel about the packet, which may already be present in the device's memory. The kernel then checks the software interrupt scheduled by schednetisr to process the packets on the IP input queue, a tasklet for the ARP input queue.

## ether\_output Function

We now examine the output of Ethernet frames. When IP calls the if\_output function, specified in the `if.h` header, the function for all Ethernet devices is ether\_output ([Figure 4.15](#)). This function creates an Ethernet frame, encapsulates it with the 14-byte IEEE 802.3 header, and sends it to the send queue. This is a large function so we describe its main steps:

- verification,

- protocol-specific processing,
- frame construction, and
- interface queueing.

[Figure 4.15](#) includes the first part of the function

### **Figure 4.15. ether\_out()**

---

```

49 int
50 ether_output(ifp, m0, dst, rt0)
51 struct ifnet *ifp;
52 struct mbuf *m0;
53 struct sockaddr *dst;
54 struct rtentry *rt0;
55 {
56     short    type;
57     int      s, error = 0;
58     u_char   edst[6];
59     struct mbuf *m = m0;
60     struct rtentry *rt;
61     struct mbuf *mcopy = (struct mbuf *) 0;
62     struct ether_header *eh;
63     int      off, len = m->m_pkthdr.len;
64     struct arpcom *ac = (struct arpcom *) ifp;

65     if ((ifp->if_flags & (IFF_UP | IFF_RUNNING)) != (IFF_UP | IFF_RUNNING))
66         senderr(ENETDOWN);
67     ifp->if_lastchange = time;
68     if (rt = rt0) {
69         if ((rt->rt_flags & RTF_UP) == 0) {
70             if (rt0 = rt = rtalloc1(dst, 1))
71                 rt->rt_refcnt--;
72             else
73                 senderr(EHOSTUNREACH);
74         }
75         if (rt->rt_flags & RTF_GATEWAY) {
76             if (rt->rt_gwroute == 0)
77                 goto lookup;
78             if (((rt = rt->rt_gwroute)->rt_flags & RTF_UP) == 0) {
79                 rtfree(rt);
80                 rt = rt0;
81             lookup:    rt->rt_gwroute = rtalloc1(rt->rt_gateway, 1);
82                 if ((rt = rt->rt_gwroute) == 0)
83                     senderr(EHOSTUNREACH);
84             }
85         }
86         if (rt->rt_flags & RTF_REJECT)
87             if (rt->rt_rmx.rmx_expire == 0 ||
88                 time.tv_sec < rt->rt_rmx.rmx_expire)
89                 senderr(rt == rt0 ? EHOSTDOWN : EHOSTUNREACH);
90     }

```

---

if\_ETHERSUBR.C

## 49-64

The arguments to ether\_output are ifp, which points to m0, the packet to send; dst, the destination address.

## 65-67

The macro senderr is called throughout ether\_output.

```
#define senderr(e) { error = (e)
```

senderr saves the error code and jumps to bad discarded and ether\_output returns error.

If the interface is up and running, ether\_output Otherwise, it returns ENETDOWN.

## Host route

68-74

rt0 points to the routing entry located by ip\_out called from BPF, rt0 can be null, in which case ENETDOWN is returned. Otherwise, the route is verified. If the route is invalid, EHOSTUNREACH is returned if a route cannot be found for the next-hop destination.

**Figure 4.16. ether\_output function**

```

91     switch (dst->sa_family) {
92     case AF_INET:
93         if (!arpresolve(ac, rt, m, dst, edst))
94             return (0); /* if not yet resolved */
95         /* If broadcasting on a simplex interface, loopback a copy */
96         if ((m->m_flags & M_BCAST) && (ifp->if_flags & IFF_SIMPLEX))
97             mcopy = m_copy(m, 0, (int) M_COPYALL);
98         off = m->m_pkthdr.len - m->m_len;
99         type = ETHERTYPE_IP;
100        break;
101    case AF_ISO:
102        /* ISO code */
103
142    case AF_UNSPEC:
143        eh = (struct ether_header *) dst->sa_data;
144        bcopy((caddr_t) eh->ether_dhost, (caddr_t) edst, sizeof(edst));
145        type = eh->ether_type;
146        break;
147    default:
148        printf("%s%d: can't handle af%d\n", ifp->if_name, ifp->if_unit,
149              dst->sa_family);
150        senderr(EAFNOSUPPORT);
151    }

```

*if\_ETHERSUBR.C*

## Gateway route

75-85

If the next hop for the packet is a gateway (vei located and pointed to by rt. If a gateway route this point, rt points to the route for the next-hc the final destination.

## Avoid ARP flooding

86-90

The RTF\_REJECT flag is enabled by the ARP coc

destination is not responding to ARP requests.

ether\_output processing continues according to Ethernet devices respond only to Ethernet address that corresponds to the IP address (Chapter 21) implements this translation. Figure 4.16 illustrates this protocol.

## IP output

91-101

ether\_output jumps according to sa\_family in the AF\_INET, AF\_UNIX, AF\_ISO, and AF\_UNSPEC cases in Figure 4.16.

The AF\_INET case calls arpresolve to determine the destination IP address. If the Ethernet address is not found, ether\_output proceeds. Otherwise this IP address, it calls ether\_output from the function

Assuming the ARP cache contains the hardware address going to be broadcast and if the interface is suitable for both tests are true, m\_copy makes a copy of the packet if it had arrived on the Ethernet interface. This is because the sending host must receive a copy of the packet.

We'll see in Chapter 12 that multicast packets are sent through the broadcast output interface.

## Explicit Ethernet output

142-146

Some protocols, such as ARP, need to specify the address family constant AF\_UNSPEC. This indicates that the caller does not care about the type of interface used to transmit the packet. The kernel then duplicates the destination address in edst and calls arprelax to call arpresolve (as for AF\_INET) because the caller did not specify an interface explicitly by the caller.

## Unrecognized address families

147-151

Unrecognized address families generate a consistent error code EAFNOSUPPORT.

In the next section of ether\_output, shown in Figure 4.17, we will see how the kernel handles unrecognized address families.

**Figure 4.17. ether\_output function**

```

152     if (mcopy)
153         (void) looutput(ifp, mcopy, dst, rt);
154     /*
155      * Add local net header.  If no space in first mbuf,
156      * allocate another.
157     */
158     M_PREPEND(m, sizeof(struct ether_header), M_DONTWAIT);
159     if (m == 0)
160         senderr(ENOBUFS);
161     eh = mtod(m, struct ether_header *);
162     type = htons((u_short) type);
163     bcopy((caddr_t) &type, (caddr_t) &eh->ether_type,
164           sizeof(eh->ether_type));
165     bcopy((caddr_t)edst, (caddr_t)eh->ether_dhost, sizeof(edst));
166     bcopy((caddr_t)ac->ac_enaddr, (caddr_t)eh->ether_shost,
167           sizeof(eh->ether_shost));

```

if\_ETHERSUBR.C

## Ethernet header

152-167

If the code in the switch made a copy of the packet received on the output interface by calling `looutput` described in [Section 5.4](#).

`M_PREPEND` ensures that there is room for 14 bytes.

Most protocols arrange to leave room at the front of the packet only to adjust some pointers (e.g., `sosend` for `inetd` described in [Section 13.6](#)).

`ether_output` forms the Ethernet header from the information provided. The source address is the unicast Ethernet address associated with the interface. The destination address for all frames transmitted on the interface is the unicast Ethernet address the caller may have specified in the `ether_header`. It is difficult to forge the source address of an Ethernet frame.

At this point, the mbuf contains a complete Eth computed by the Ethernet hardware during trai the frame for transmission by the device.

**Figure 4.18. ether\_output**

```
if_etheroutput.c
168     s = splimp();
169     /*
170      * Queue message on interface, and start output if interface
171      * not yet active.
172      */
173     if (IF_QFULL(&ifp->if_snd)) {
174         IF_DROP(&ifp->if_snd);
175         splx(s);
176         senderr(ENOBUFS);
177     }
178     IF_ENQUEUE(&ifp->if_snd, m);
179     if ((ifp->if_flags & IFF_OACTIVE) == 0)
180         (*ifp->if_start) (ifp);
181     splx(s);
182     ifp->if_cbytes += len + sizeof(struct ether_header);
183     if (m->m_flags & M_MCAST)
184         ifp->if_omcasts++;
185     return (error);

186     bad:
187     if (m)
188         m_freem(m);
189     return (error);
190 }
```

168-185

If the output queue is full, ether\_output discards the frame. If the output queue is not full, the frame is placed on the interface's output queue. The if\_start function transmits the next frame if the interface is not already active.

186-190

The senderr macro jumps to bad where the frame is discarded.

## lrestart Function

The lrestart function dequeues frames from the transmitted by the LANCE Ethernet card. If the transmitting frames. An example appears at the is called indirectly through the interface's if\_start.

If the device is busy, it generates an interrupt via frame. The driver calls lrestart to dequeue and the layer can queue frames without calling lrestart so the queue is empty.

Figure 4.19 shows the lrestart function. lrestart handles interrupts.

**Figure 4.19.**

```

325 lestart(ifp)                                if_le.c
326 struct ifnet *ifp;
327 {
328     struct le_softc *le = &le_softc;ifp->if_unit);
329     struct letmd *tmd;
330     struct mbuf *m;
331     int len;
332     if ((le->sc_if.if_flags & IFF_RUNNING) == 0)
333         return (0);

334                                         /* device-specific code */

335     do {

336                                         /* device-specific code */

340     IF_DEQUEUE(&le->sc_if.if_snd, m);
341     if (m == 0)
342         return (0);
343     len = leput(le->sc_r2->ler2_tbuf[le->sc_tmd], m);
344     /*
345      * If bpf is listening on this interface, let it
346      * see the packet before we commit it to the wire.
347      */
348     if (ifp->if_bpf)
349         bpf_tap(ifp->if_bpf, le->sc_r2->ler2_tbuf[le->sc_tmd],
350                 len);

354                                         /* device-specific code */

359     } while (++le->sc_txcnt < LETBUF);
360     le->sc_if.if_flags |= IFF_OACTIVE;
361     return (0);
362 }

```

if\_le.c

## Interface must be initialized

325-333

If the interface is not initialized, lestart returns

## Dequeue frame from output queue

335-342

If the interface is initialized, the next frame is ready. If the queue is empty, leststart returns.

## **Transmit frame and pass to BPF**

343-350

Ieput copies the frame in m to the hardware buffer. If the interface is tapped by BPF, the frame is passed to the BPF code that initiates the transmission of the frame.

## **Repeat if device is ready for more frames**

359

leststart stops passing frames to the device when the number of outgoing interfaces can queue more than one. E.g. if there are two outgoing interfaces and only one number of hardware transmit buffers available, then both of the buffers are in use.

## **Mark device as busy**

360-362

Finally, leststart turns on IFF\_OACTIVE in the ifnet structure for transmitting frames.

There is an unfortunate side effect to queuing in the driver. According to [Jacobson 1988a], the LANCE chip can't handle little delay between frames. Unfortunately, so because they can't process the incoming data

This interacts badly with an application such as NFS (files greater than 8192 bytes) that are fragmented into multiple Ethernet frames. Fragments are lost due to incomplete datagrams and high delays as NFS reassembles them.

Jacobson noted that Sun's LANCE driver only had this problem.

## Chapter 4. Interfaces: Ethernet

---

### 4.4 ioctl System Call

The ioctl system call supports a generic command for a device that aren't supported by the standard system calls.

```
int ioctl (int fd, unsigned long cmd, ...);
```

*fd* is a descriptor, usually a device or network connection. *cmd* is a command of ioctl commands specified by the second argument. The third argument is a pointer to some type that depends on the command. If the command is retrieving information, the third argument points to a buffer for the data. In this text, we discuss only the ioctl command for retrieving information.

The prototype we show for system calls is the same as shown in Chapter 15. Note that the function within the prototype is not the actual system call, but rather a wrapper function that performs the necessary setup before calling the system call.

We describe the implementation of the ioctl system call in Chapter 15. We also provide an implementation of individual ioctl commands that are used in the chapter.

The first ioctl commands we discuss provide access to interface configuration as described. Throughout the text we summarize important ioctl commands.

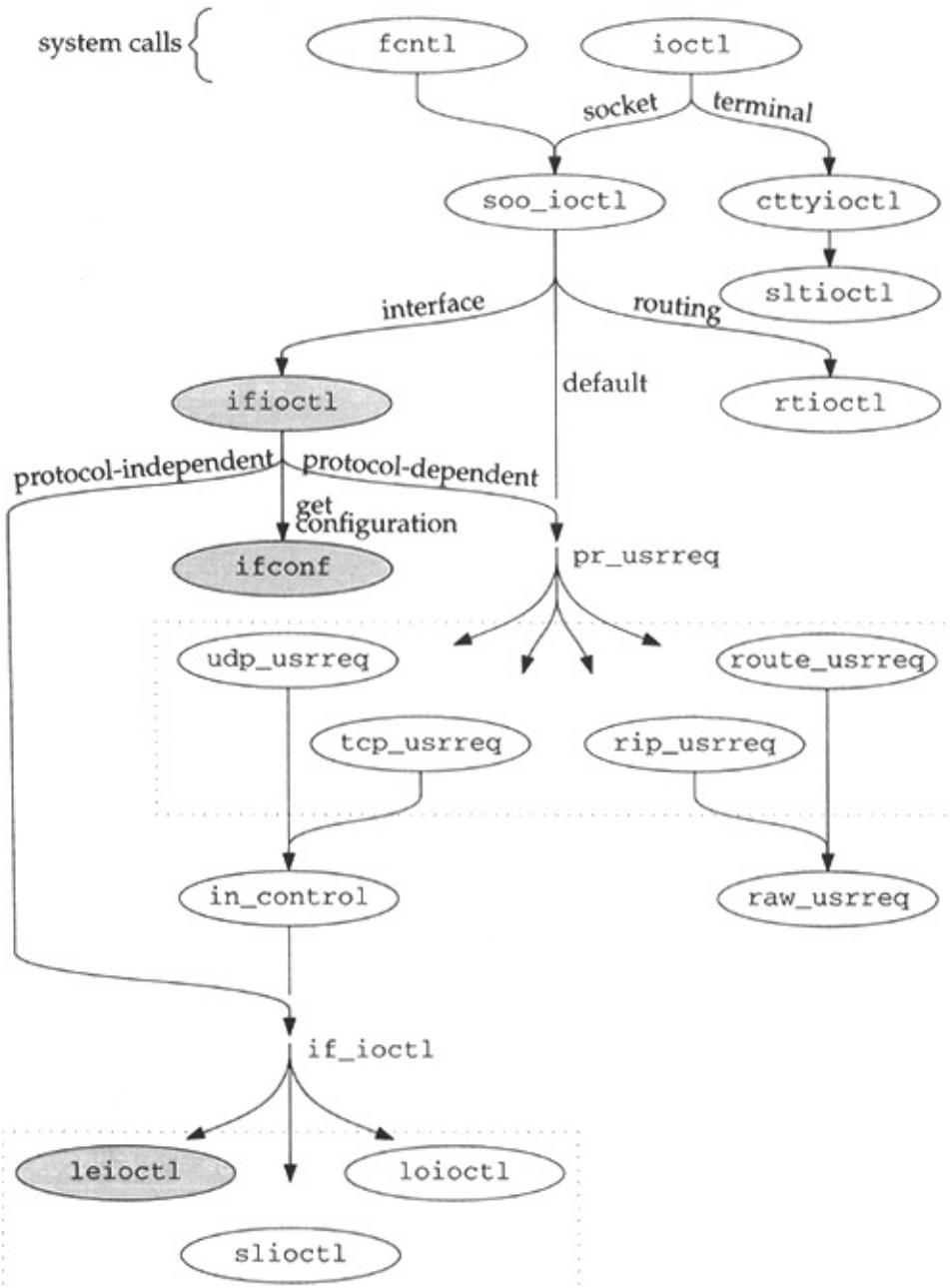
**Figure 4.20. Interface ioctl commands**

Command	Third argument	Function	Description
<i>SIOCGIFCONF</i>	struct ifconf *	ifconf	retrieve list of interface configuration
<i>SIOCGIFFLAGS</i>	struct ifreq *	ifioctl	get interface flags
<i>SIOCGIFMETRIC</i>	struct ifreq *	ifioctl	get interface metric
<i>SIOCSIFFLAGS</i>	struct ifreq *	ifioctl	set interface flags
<i>SIOCSIFMETRIC</i>	struct ifreq *	ifioctl	set interface metric

The first column shows the symbolic constant to use with the ioctl command (*com*). The second column shows the type of the third argument for the command shown in the first column. The third column shows the function name for the command.

Figure 4.21 shows the organization of the various ioctl functions. The functions are the ones we describe in this chapter and the chapters that follow.

**Figure 4.21. ioctl functions**



## ifioctl Function

The ioctl system call routes the five commands  
Figure 4.22.

## Figure 4.22. ifioctl funct

```
if.c  
394 int  
395 ifioctl(so, cmd, data, p)  
396 struct socket *so;  
397 int cmd;  
398 caddr_t data;  
399 struct proc *p;  
400 {  
401     struct ifnet *ifp;  
402     struct ifreq *ifr;  
403     int error;  
404     if (cmd == SIOCGIFCONF)  
405         return (ifconf(cmd, data));  
406     ifr = (struct ifreq *) data;  
407     ifp = ifunit(ifr->ifr_name);  
408     if (ifp == 0)  
409         return (ENXIO);  
410     switch (cmd) {  
  
        /* other interface ioctl commands (Figures 4.29 and 12.11) */  
  
447     default:  
448         if (so->so_proto == 0)  
449             return (EOPNOTSUPP);  
450         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,  
451                                         cmd, data, ifp));  
452     }  
453     return (0);  
454 }
```

if.c

394-405

For the SIOCGIFCONF command, ifioctl calls ifc structures.

406-410

For the remaining ioctl commands, the data arg is the ifnet list for an interface with the text name ("sl0", "le1", or "lo0"). If there is no matching interface, the function depends on cmd and is described with Figure 4.

447-454

If the interface ioctl command is not recognized function of the protocol associated with the socket commands are issued on a UDP socket and udp category are described in [Figure 6.10](#). Section :

If control falls out of the switch, 0 is returned.

## ifconf Function

ifconf provides a standard way for a process to configured on a system. Interface information is shown in Figures 4.23 and 4.24.

Figure 4.

```
-----if.h-----  
262 struct ifreq {  
263 #define IFNAMSIZ 16  
264     char ifr_name[IFNAMSIZ];           /* if name, e.g. "en0" */  
265     union {  
266         struct sockaddr ifru_addr;  
267         struct sockaddr ifru_dstaddr;  
268         struct sockaddr ifru_broadaddr;  
269         short ifru_flags;  
270         int ifru_metric;  
271         caddr_t ifru_data;  
272     } ifr_ifru;  
273 #define ifr_addr ifr_ifru.ifru_addr      /* address */  
274 #define ifr_dstaddr ifr_ifru.ifru_dstaddr /* other end of p-to-p link */  
275 #define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */  
276 #define ifr_flags ifr_ifru.ifru_flags    /* flags */  
277 #define ifr_metric ifr_ifru.ifru_metric  /* metric */  
278 #define ifr_data ifr_ifru.ifru_data    /* for use by interface */  
279 };  
-----if.h-----
```

**Figure 4.1**

```
-----if.h-----  
292 struct ifconf {  
293     int ifc_len;           /* size of associated buffer */  
294     union {  
295         caddr_t ifcu_buf;  
296         struct ifreq *ifcu_req;  
297     } ifc_ifcu;  
298 #define ifc_buf ifc_ifcu.ifcu_buf    /* buffer address */  
299 #define ifc_req ifc_ifcu.ifcu_req   /* array of structures returned */  
300 };  
-----if.h-----
```

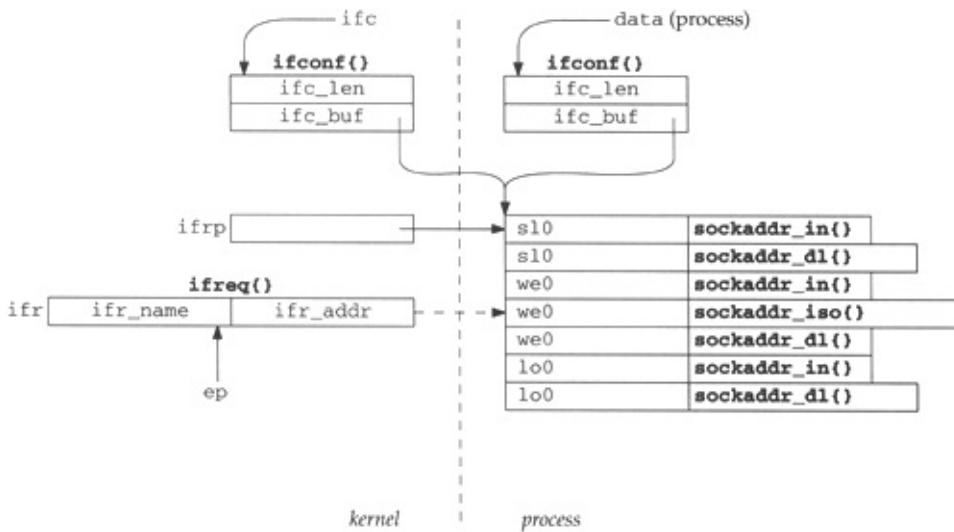
**262-279**

An ifreq structure contains the name of an interface and its flags. These are accessed by the various ioctl commands. As shown in the code, ifc\_ifcu is a union containing two members: ifcu\_buf and ifcu\_req.

**292-300**

In the ifconf structure, ifc\_len is the size in bytes of the buffer allocated by a process but filled in by ifconf with data from the ifconfig function, ifr\_addr is the relevant member of the ifreq structure. The ifr\_ifcu member has a variable length because the length of ifr\_ifcu depends on the length of the address. The sa\_len member from the sockaddr structure is used to indicate the length of the address. The sa\_data member from the sockaddr structure is used to store the address. The ifr\_ifcu member is a union containing two members: ifcu\_buf and ifcu\_req.

**Figure 4.25.**



In Figure 4.25, the data on the left is in the kernel and the data on the right is in the process. This diagram is similar to Figure 4.25, but it shows the data structures in the kernel and process separately. In this diagram, the data on the left is in the kernel and the data on the right is in the process. This diagram is similar to Figure 4.25, but it shows the data structures in the kernel and process separately.

Figure 4.

---

```

462 int
463 ifconf(cmd, data)
464 int      cmd;
465 caddr_t data;
466 {
467     struct ifconf *ifc = (struct ifconf *) data;
468     struct ifnet *ifp = ifnet;
469     struct ifaddr *ifa;
470     char    *cp, *ep;
471     struct ifreq ifr, *ifrp;
472     int      space = ifc->ifc_len, error = 0;

```

```

473     ifrp = ifc->ifc_req;
474     ep = ifr.ifr_name + sizeof(ifr.ifr_name) - 2;
475     for (; space > sizeof(ifr) && ifp; ifp = ifp->if_next) {
476         strncpy(ifr.ifr_name, ifp->if_name, sizeof(ifr.ifr_name) - 2);
477         for (cp = ifr.ifr_name; cp < ep && *cp && cp++;)
478             continue;
479         *cp++ = '0' + ifp->if_unit;
480         *cp = '\0';
481         if ((ifa = ifp->if_addrlist) == 0) {
482             bzero((caddr_t) & ifr.ifr_addr, sizeof(ifr.ifr_addr));
483             error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
484                             sizeof(ifr));
485             if (error)
486                 break;
487             space -= sizeof(ifr), ifrp++;
488         } else
489             for (; space > sizeof(ifr) && ifa; ifa = ifa->ifa_next) {
490                 struct sockaddr *sa = ifa->ifa_addr;
491                 if (sa->sa_len <= sizeof(*sa)) {
492                     ifr.ifr_addr = *sa;
493                     error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
494                                     sizeof(ifr));
495                     ifrp++;
496                 } else {
497                     space -= sa->sa_len - sizeof(*sa);
498                     if (space < sizeof(ifr))
499                         break;
500                     error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
501                                     sizeof(ifr.ifr_name));
502                     if (error == 0)
503                         error = copyout((caddr_t) sa,
504                                         (caddr_t) & ifrp->ifr_addr, sa->sa_len);
505                     ifrp = (struct ifreq *)
506                           (sa->sa_len + (caddr_t) & ifrp->ifr_addr);
507                 }
508                 if (error)
509                     break;
510                 space -= sizeof(ifr);
511             }
512         }
513     ifc->ifc_len -= space;
514     return (error);
515 }

```

if.c

## 462-474

The two arguments to ifconf are: cmd, which is structure specified by the process.

ifc is data cast to a ifconf structure pointer. ifp (the list), and ifa traverses the address list for each interface name within ifr, which is the ifreq before they are copied to the process's buffer. ifr's address is copied. space is the number of bytes

for the end of the name, and ep marks the last name.

## 475-488

The for loop traverses the list of interfaces. For followed by the text representation of the if\_un interface, an address of all 0s is constructed, tr is decreased, and ifrp is advanced.

## 489-515

If the interface has one or more addresses, the the interface name in ifr and then ifr is copied to sockaddr structure don't fit in ifr and are copied and ifrp are adjusted. After all the interfaces are >ifc\_len) and ifconf returns. The ioctl system call structure back to the ifconf structure in the program.

## Example

[Figure 4.27](#) shows the configuration of the interfaces after all the interfaces have been initialized.

## Figure 4.27. Interface Configuration

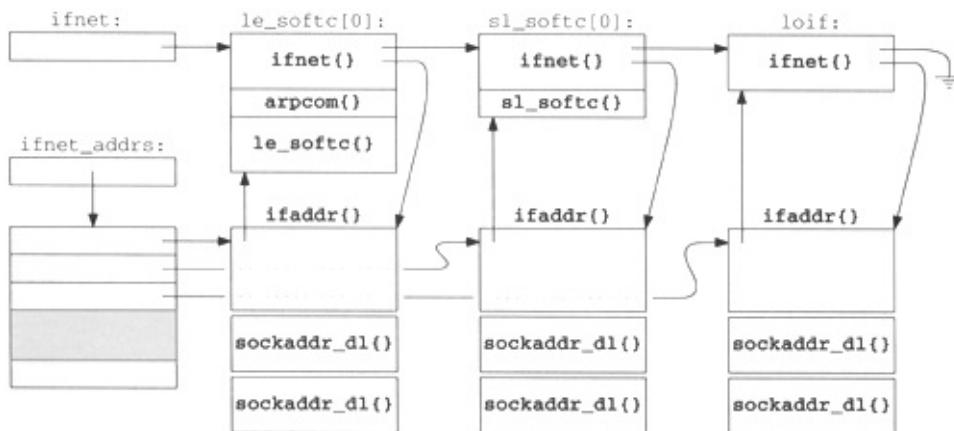
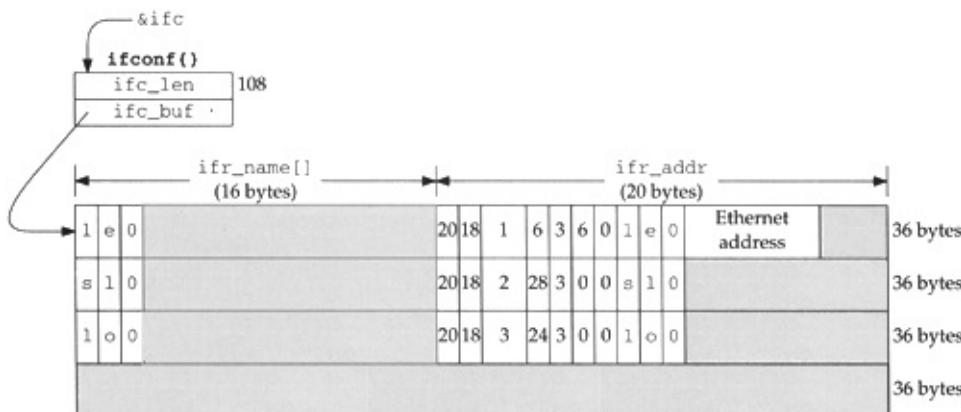


Figure 4.28 shows the contents of ifc and buffer

**Figure 4.28. Data returned by SIOCGIFCONF**



```

struct ifconf ifc;           /* * SIOCGIFC
char buffer[144];           /* contain
int s;                      /* any soc

ifc.ifc_len = 144;
ifc.ifc_buf = buffer;

```

```
    if (ioctl(s, SIOCGIFCONF, &ifc)
        perror("ioctl failed");
        exit(1);
}
```

There are no restrictions on the type of socket have seen, returns the addresses for all protocols.

In [Figure 4.28](#), ifc\_len has been changed from the buffer only occupy 108 (3x36) bytes. Three bytes of the buffer are unused. The first 16 bytes of the buffer are used. In this case only 3 of the 16 bytes are used.

ifr\_addr has the form of a sockaddr structure, so its value is the type of address (18, AF\_LINK). The values for the interface as is sdl\_type (6, 28, and 24 correspondingly).

The next three values are sa\_nlen (the length of the address), and sa\_slen (unused). sa\_nlen is 3 for both the SLIP and loopback interfaces.

Finally, the text interface name appears, followed by the interface number. Neither the SLIP nor the loopback interface store a hardware address.

In the example, only sockaddr\_dl addresses are configured (as shown in [Figure 4.27](#)), so each entry in the list (containing the OSI addresses) were configured for an interface with a single hardware address, and the size of each entry would vary.

## Generic Interface ioctl commands

The four remaining interface commands from F SIOCGIFFLAGS, and SIOCSIFMETRIC) are hand statements for these commands.

Figure 4.29. ifioctl

```
410 switch (cmd) {                                     -if.c
411     case SIOCGIFFLAGS:
412         ifr->ifr_flags = ifp->if_flags;
413         break;
414
415     case SIOCGIFMETRIC:
416         ifr->ifr_metric = ifp->if_metric;
417         break;
418
419     case SIOCSIFFLAGS:
420         if (error = suser(p->p_ucred, &p->p_acflag))
421             return (error);
422         if (ifp->if_flags & IFF_UP && (ifr->ifr_flags & IFF_UP) == 0) {
423             int      s = splimp();
424             if_down(ifp);
425             splx(s);
426         }
427         if (ifr->ifr_flags & IFF_UP && (ifp->if_flags & IFF_UP) == 0) {
428             int      s = splimp();
429             if_up(ifp);
430             splx(s);
431         }
432         ifp->if_flags = (ifp->if_flags & IFF_CANTCHANGE) |
433                         (ifr->ifr_flags & ~IFF_CANTCHANGE);
434         if (ifp->if_ioctl)
435             (*ifp->if_ioctl) (ifp, cmd, data);
436         break;
437
438     case SIOCSIMETRIC:
439         if (error = suser(p->p_ucred, &p->p_acflag))
440             return (error);
441         ifp->if_metric = ifr->ifr_metric;
442         break;
```

## SIOCGIFFLAGS and SIOCGIFMETRIC

410-416

For the two SIOCGxxx commands, ifioctl copies ifreq structure. For the flags, the ifr\_flags mem ifr\_metric member is used ([Figure 4.23](#)).

## SIOCSIFFLAGS

417-429

To change the interface flags, the calling process shutting down a running interface or bringing up an interface respectively.

## Ignore IFF\_CANTCHANGE flags

430-434

Recall from [Figure 3.7](#) that some interface flags (>if\_flags & IFF\_CANTCHANGE) clears the interface expression (ifr->ifr\_flags &~IFF\_CANTCHANGE) by the process. The two expressions are ORed together. Before returning, the request is passed to the interface driver for the LANCE driver([Figure 4.31](#)).

## SIOCSIFMETRIC

435-439

Changing the interface metric is easier; as long as we can set the new metric into if\_metric for the interface.

## if\_down and if\_up Functions

With the ifconfig program, an administrator can set the IFF\_UP flag through the SIOCSIFFLAGS command. The if\_down and if\_up functions handle these calls.

Figure 4.30. if.c

```
if.c
292 void
293 if_down(ifp)
294 struct ifnet *ifp;
295 {
296     struct ifaddr *ifa;
297     ifp->if_flags &= ~IFF_UP;
298     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
299         pfctlinput(PRC_IFDOWN, ifa->ifa_addr);
300     if_qflush(&ifp->if_snd);
301     rt_ifmsg(ifp);
302 }

308 void
309 if_up(ifp)
310 struct ifnet *ifp;
311 {
312     struct ifaddr *ifa;
313     ifp->if_flags |= IFF_UP;
314     rt_ifmsg(ifp);
315 }
```

292-302

When an interface is shut down, the IFF\_UP flag is cleared. The if\_down function sends a pfctlinput (Section 7.7) for each address associated with the interface, giving the opportunity to respond to the interface being shutdown.

connections using the interface. IP attempts to TCP and UDP ignore failing interfaces and rely on packets.

`if_qflush` discards any packets queued for the interface in `rt_ifmsg`. TCP retransmits the lost packets automatically. UDP does not respond to this condition on their own.

308-315

When an interface is enabled, the IFF\_UP flag is set. When an interface is disabled, the IFF\_UP flag is cleared. When an interface status has changed.

## Ethernet, SLIP, and Loopback

We saw in [Figure 4.29](#) that for the SIOCSIFFLAGS command, the driver ignores the SIOCSIFFLAGS command. In our three sample interfaces, the `slif_ioctl` command, which is ignored by `ifioctl`. [Figure 4.31](#) shows the processing of the LANCE Ethernet driver.

**Figure 4.31.** `leioctl`

---

```

614 leioctl(ifp, cmd, data)           if_le.c
615 struct ifnet *ifp;
616 int     cmd;
617 caddr_t data;
618 {
619     struct ifaddr *ifa = (struct ifaddr *) data;
620     struct le_softc *le = &le_softc[ifp->if_unit];
621     struct leregi *ler1 = le->sc_r1;
622     int      s = splimp(), error = 0;
623
624     switch (cmd) {

625         /* SIOCSIFADDR code (Figure 6.28) */

638     case SIOCSIFFLAGS:
639         if (((ifp->if_flags & IFF_UP) == 0 &&
640             ifp->if_flags & IFF_RUNNING) ||
641             LERDWR(le->sc_r0, LE_STOP, ler1->ler1_rdp);
642             ifp->if_flags &= ~IFF_RUNNING;
643         } else if ((ifp->if_flags & IFF_UP &&
644             (ifp->if_flags & IFF_RUNNING) == 0)
645             leinit(ifp->if_unit);
646         /*
647          * If the state of the promiscuous bit changes, the interface
648          * must be reset to effect the change.
649          */
650         if (((ifp->if_flags ^ le->sc_iflags) & IFF_PROMISC) &&
651             (ifp->if_flags & IFF_RUNNING)) {
652             le->sc_iflags = ifp->if_flags;
653             lereset(ifp->if_unit);
654             lestart(ifp);
655         }
656     break;

657         /* SIOCADDMULTI and SIOCDELMULTI code (Figure 12.31) */

672     default:
673         error = EINVAL;
674     }
675     splx(s);
676     return (error);
677 }

```

---

## 614-623

leioctl casts the third argument, data, to an ifaddr pointer references the le\_softc structure indexe makes up the main body of the function.

## 638-656

Only the SIOCSIFFLAGS case is shown in Figure 6.28. The other cases have been changed. The code shown here forces

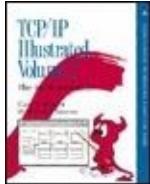
configuration of the flags. If the interface is going to be operating, the interface is shut down. If the interface is not initialized and restarted.

If the promiscuous bit has been changed, the interface is reinitialized after the change.

The expression including the exclusive OR and the IFF\_PROMISC bit.

672-677

The default case for unrecognized commands performs the function.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 4. Interfaces: Ethernet

### 4.5 Summary

In this chapter we described the implementation of the LANCE Ethernet device driver, which we refer to throughout the text. We saw how the Ethernet driver detects broadcast and multicast addresses on input, how the Ethernet and 802.3 encapsulations are detected, and how incoming frames are demultiplexed to the appropriate protocol queue. In [Chapter 21](#) we'll see how IP addresses (unicast, broadcast, and multicast) are converted into the correct Ethernet addresses on output.

Finally, we discussed the protocol-specific ioctl commands that access the interface-

layer data structures.

## Exercises

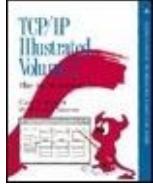
In leread, the M\_MCAST flag (in addition to M\_BCAST) is always set when a broadcast packet is received.

**4.1** Compare this behavior to the code in ether\_input. Why are the flags set in leread and ether\_input? Does it matter? Which is correct?

In ether\_input (Figure 4.13), what would happen if the test for the broadcast address and the test for a

**4.2** multicast address were swapped? What would happen if the if on the test for a multicast address were not preceded by an else?





[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Chapter 5. Interfaces: SLIP and Loopback

[Section 5.1. Introduction](#)

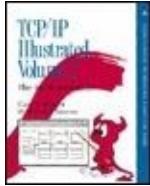
[Section 5.2. Code Introduction](#)

[Section 5.3. SLIP Interface](#)

[Section 5.4. Loopback Interface](#)

[Section 5.5. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 5. Interfaces: SLIP and Loopback

### 5.1 Introduction

In [Chapter 4](#) we looked at the Ethernet interface. In this chapter we describe the SLIP and loopback interfaces, as well as the ioctl commands used to configure all network interfaces. The TCP compression algorithm used by the SLIP driver is described in [Section 29.13](#). The loopback driver is straightforward and we discuss it here in its entirety.

[Figure 5.1](#), which also appeared as [Figure 4.2](#), lists the entry points to our three example drivers.

## Figure 5.1. Interface functions for the example drivers.

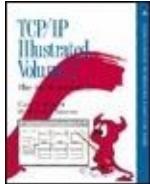
ifnet	Ethernet	SLIP	Loopback	Description
if_init	leinit			initialize hardware
if_output	ether_output	sloutput	looutput	accept and queue packet for transmission
if_start	lestart			begin transmission of frame
if_done				output complete (unused)
if_ioctl	leioctl	sliioctl	loioctl	handle ioctl commands from a process
if_reset	lereset			reset the device to a known state
if_watchdog				watch the device for failures or collect statistics

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 5. Interfaces: SLIP and Loopback

### 5.2 Code Introduction

The files containing code for SLIP and loopback drivers are listed in [Figure 5.2](#).

**Figure 5.2. Files discussed in this chapter.**

File	Description
net/if_slvar.h	SLIP definitions
net/if_sl.c	SLIP driver functions
net/if_loop.c	loopback driver

### Global Variables

The SLIP and loopback interface structures

are described in this chapter.

### Figure 5.3. Global variables introduced in this chapter.

Variable	Datatype	Description
sl_softc	struct sl_softc []	SLIP interface
loif	struct ifnet	loopback interface

sl\_softc is an array, since there can be many SLIP interfaces. loif is not an array, since there can be only one loopback interface.

### Statistics

The statistics from the ifnet structure described in [Chapter 4](#) are also updated by the SLIP and loopback drivers. One other variable (which is not in the ifnet structure) collects statistics; it is shown in [Figure 5.4](#).

### Figure 5.4. tk\_nin variable.

Variable	Description	Used by SNMP
tk_nin	#bytes received by any serial interface (updated by SLIP driver)	

---

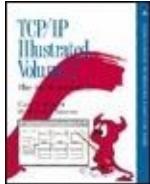
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



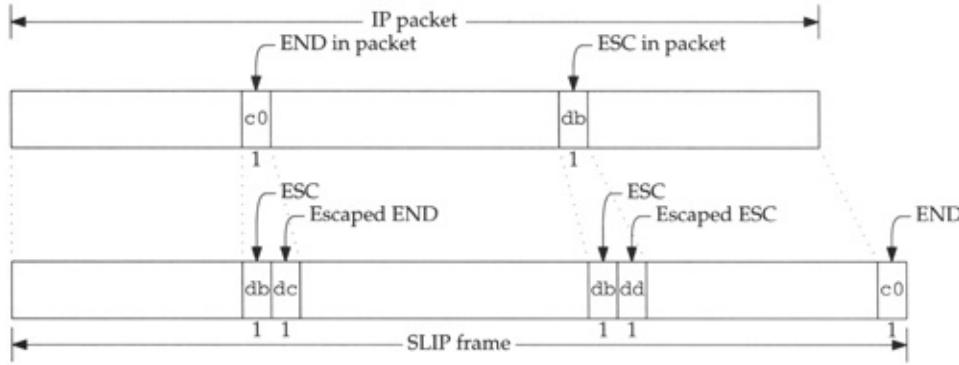
[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 5. Interfaces: SLIP and Loopback

### 5.3 SLIP Interface

A SLIP interface communicates with a remote system across a standard asynchronous serial line. As with Ethernet, SLIP defines a standard way to frame IP packets as they are transmitted on the serial line. [Figure 5.5](#) shows the encapsulation of an IP packet into a SLIP frame when the IP packet contains SLIP's reserved characters.

**Figure 5.5. SLIP encapsulation of an IP packet.**



Packets are separated by the SLIP END character 0xc0. If the END character appears in the IP packet, it is prefixed with the SLIP ESC character 0xdb and transmitted as 0xdc instead. When the ESC character appears in the IP packet, it is prefixed with the ESC character 0xdb and transmitted as 0xdd.

Since there is no type field in SLIP frames (as there is with Ethernet), SLIP is suitable only for carrying IP packets.

SLIP is described in RFC 1055 [[Romkey 1988](#)], where its many weaknesses and nonstandard status are also stated. Volume 1 contains a more detailed description of SLIP encapsulation.

The Point-to-Point Protocol (PPP) was designed to address SLIP'S problems

and to provide a standard method for transmitting frames across a serial link. PPP is defined in RFC 1332 [[McGregor 1992](#)] and RFC 1548 [[Simpson 1993](#)]. Net/3 does not contain an implementation of PPP, so we do not discuss it in this text. See Section 2.6 of Volume 1 for more information regarding PPP. [Appendix B](#) describes where to obtain a reference implementation of PPP.

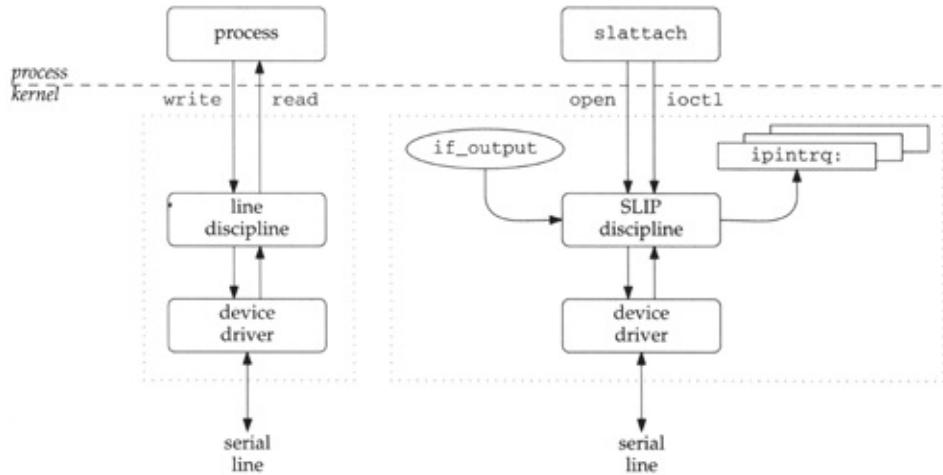
## The SLIP Line Discipline: `slipdisc`

In Net/3 the SLIP interface relies on an asynchronous serial device driver to send and receive the data. Traditionally these device drivers have been called TTYs (teletypes). The Net/3 TTY subsystem includes the notion of a *line discipline* that acts as a filter between the physical device and I/O system calls such as read and write. A line discipline implements features such as line editing, newline and carriage-return processing, tab expansion, and more. The SLIP interface appears as a line discipline to the TTY subsystem, but it

does not pass incoming data to a process reading from the device and does not accept outgoing data from a process writing to the device. Instead, the SLIP interface passes incoming packets to the IP input queue and accepts outgoing packets through the `if_output` function in SLIP's `ifnet` structure. The kernel identifies line disciplines by an integer constant, which for SLIP is `SLIPDISC`.

[Figure 5.6](#) shows a traditional line discipline on the left and the SLIP discipline on the right. We show the process on the right as `slattach` since it is the program that initializes a SLIP interface. The details of the TTY subsystem and line disciplines are outside the scope of this text. We present only the information required to understand the workings of the SLIP code. For more information about the TTY subsystem see [[Leffler et al. 1989](#)]. [Figure 5.7](#) lists the functions that implement the SLIP driver. The middle columns indicate whether the function implements line discipline features, network interface features, or both.

**Figure 5.6. The SLIP interface as a line discipline.**



**Figure 5.7. The functions in the SLIP device driver.**

Function	Network Interface	Line Discipline	Description
slattach	•		initialize and attach sl_softc structures to ifnet list
sllinit	•		initialize the SLIP data structures
sloutput	•		queue outgoing packets for transmission on associated TTY device
slioctl	•		process socket ioctl requests
sl_btom	•		convert a device buffer to an mbuf chain
slopen		•	attach sl_softc structure to TTY device and initialize driver
slclose		•	detach sl_softc structure from TTY device, mark interface as down, and release memory
sltioctl		•	process TTY ioctl commands
slstart	•	•	dequeue packet and begin transmitting data on TTY device
slinput	•	•	process incoming byte from TTY device, queue incoming packet if an entire frame has been received

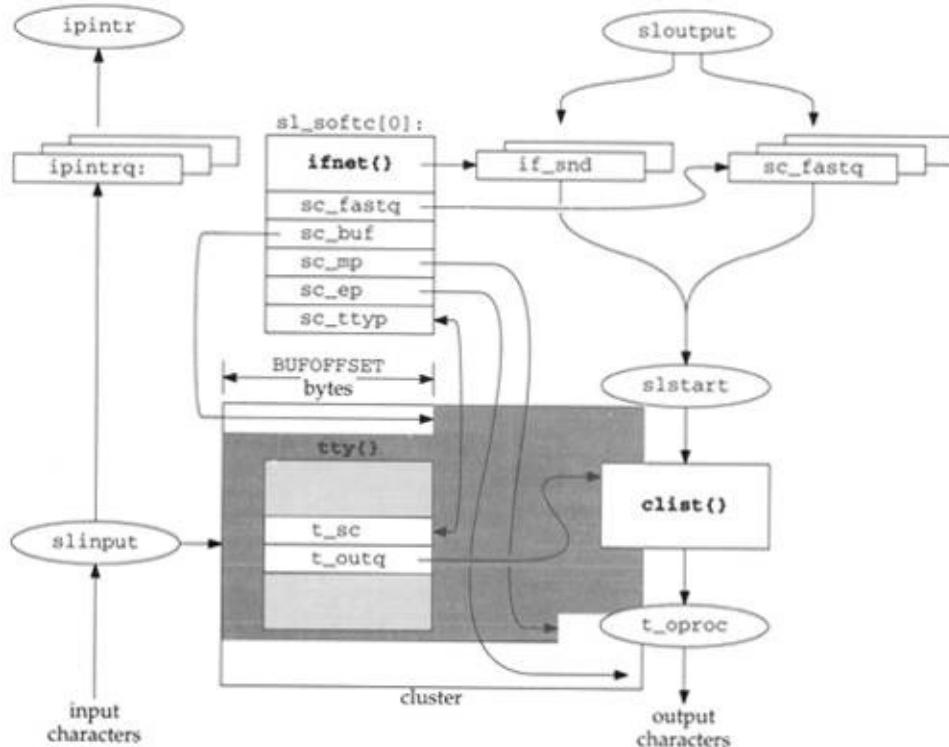
The SLIP driver in Net/3 supports compression of TCP packet headers for better throughput. We discuss header

compression in [Section 29.13](#), so [Figure 5.7](#) omits the functions that implement this feature.

The Net/3 SLIP interface also supports an escape sequence. When detected by the receiver, the sequence shuts down SLIP processing and returns the device to the standard line discipline. We omit this processing from our discussion.

[Figure 5.8](#) shows the complex relationship between SLIP as a line discipline and SLIP as a network interface.

**Figure 5.8. SLIP device driver.**



In Net/3 **sc\_ttyp** and **t\_sc** point to the **tty** structure and the **sl\_softc[0]** structure respectively. Instead of cluttering the figure with two arrows, we use a double-ended arrow positioned at each pointer to illustrated the two links between the structures.

Figure 5.8 contains a lot of information:

- The network interface is represented by the **sl\_softc** structure and the TTY device by the **tty** structure.
- Incoming bytes are stored in the cluster

(shown behind the tty structure). When a complete SLIP frame is received, the enclosed IP packet is put on the ipintrq by slinput.

- Outgoing packets are dequeued from if\_snd or sc\_fastq, converted to SLIP frames, and passed to the TTY device by slstart. The TTY buffers outgoing bytes in the clist structure. The t\_oproc function drains and transmits the bytes held in the clist structure.

## SLIP Initialization: slopen and slinit

We discussed in [Section 3.7](#) how slattach initializes the sl\_softc structures. The interface remains initialized but inoperative until a program (usually slattach) opens a TTY device (e.g., /dev/tty01) and issues an ioctl command to replace the standard line discipline with the SLIP discipline. At this point the TTY subsystem calls the line discipline's open function (in this case slopen), which establishes the association between a particular TTY device and a particular SLIP interface. slopen is shown

in Figure 5.9.

## Figure 5.9. The sopen function.

```
-----if_sl.c
181 int
182 sopen(dev, tp)
183 dev_t    dev;
184 struct tty *tp;
185 {
186     struct proc *p = curproc; /* XXX */
187     struct sl_softc *sc;
188     int      nsl;
189     int      error;
190
191     if (error = suser(p->p_ucred, &p->p_acflag))
192         return (error);
193
194     if (tp->t_line == SLIPDISC)
195         return (0);
196
197     for (nsl = NSL, sc = sl_softc; --nsl >= 0; sc++)
198         if (sc->sc_ttyp == NULL) {
199             if (slinit(sc) == 0)
200                 return (ENOBUFS);
201             tp->t_sc = (caddr_t) sc;
202             sc->sc_ttyp = tp;
203             sc->sc_if.if_baudrate = tp->t_ospeed;
204             ttyflush(tp, FREAD | FWRITE);
205         }
206     return (ENXIO);
207 }
```

-----if\_sl.c

181-193

Two arguments are passed to sopen: dev, a kernel device identifier that sopen does not use; and tp, a pointer to the tty structure associated with the TTY device. First some precautions: if the process does not have superuser privileges, or if the TTY's line discipline is set to SLIPDISC already, sopen returns immediately.

## 194-205

The for loop searches the array of sl\_softc structures for the first unused entry, calls slinit (Figure 5.10), joins the tty and sl\_softc structures by t\_sc and sc\_ttyp, and copies the TTY output speed (t\_ospeed) into the SLIP interface. ttyflush discards any pending input or output data in the TTY queues. slopen returns ENXIO if a SLIP interface structure is not available, or 0 if it was successful.

### Figure 5.10. The slinit function.

```
156 static int  
157 slinit(sc)  
158 struct sl_softc *sc;  
159 {  
160     caddr_t p;  
161     if (sc->sc_ep == (u_char *) 0) {  
162         MCLALLOC(p, M_WAIT);  
163         if (p)  
164             sc->sc_ep = (u_char *) p + SLBUFSIZE;  
165         else {  
166             printf("sl%d: can't allocate buffer\n", sc - sl_softc);  
167             sc->sc_if.if_flags &= ~IFF_UP;  
168             return (0);  
169         }  
170     }  
171     sc->sc_buf = sc->sc_ep - SLMAX;  
172     sc->sc_mp = sc->sc_buf;  
173     sl_compress_init(&sc->sc_comp);  
174     return (1);  
175 }
```

Notice that the first available sl\_softc structure is associated with the TTY

device. There need not be a fixed mapping between TTY devices and SLIP interfaces if the system has more than one SLIP line. In fact, the mapping depends on the order in which slattach opens and closes the TTY devices.

The slinit function shown in [Figure 5.10](#) initializes the sl\_softc structure.

156-175

The slinit function allocates an mbuf cluster and attaches it to the sl\_softc structure with three pointers. Incoming bytes are stored in the cluster until an entire SLIP frame has been received. sc\_buf always points to the start of the packet in the cluster, sc\_mp points to the location of the next byte to be received, and sc\_ep points to the end of the cluster. sl\_compress\_init initializes the TCP header compression state for this link ([Section 29.13](#)).

In [Figure 5.8](#) we see that sc\_buf does not point to the first byte in the cluster. slinit leaves room for 148 bytes (BUFOFFSET),

as the incoming packet may have a compressed header that will expand to fill this space. The bytes that have already been received are shaded in the cluster. We see that `sc_mp` points to the byte just after the last byte received and `sc_ep` points to the end of the cluster. [Figure 5.11](#) shows the relationships between several SLIP constants.

## Figure 5.11. SLIP constants.

Constant	Value	Description
<code>MCLBYTES</code>	2048	size of an mbuf cluster
<code>SLBUFSIZE</code>	2048	maximum size of an uncompressed SLIP packet—including a BPF header
<code>SLIP_HDRLEN</code>	16	size of SLIP BPF header
<code>BUFOFFSET</code>	148	maximum size of an expanded TCP/IP header plus room for a BPF header
<code>SLMAX</code>	1900	maximum size of a compressed SLIP packet stored in a cluster
<code>SLMTU</code>	296	optimal size of SLIP packet; results in minimal delay with good bulk throughput
<code>SLIP_HIWAT</code>	100	maximum number of bytes to queue in TTY output queue
$\text{BUFOFFSET} + \text{SLMAX} = \text{SLBUFSIZE} = \text{MCLBYTES}$		

All that remains to make the interface operational is to assign it an IP address. As with the Ethernet driver, we postpone the discussion of address assignment until [Section 6.6](#).

## SLIP Input Processing: slinput

The TTY device driver delivers incoming characters to the SLIP line discipline one at a time by calling `slinput`. [Figure 5.12](#) shows the `slinput` function but omits the end-of-frame processing, which is discussed separately.

**Figure 5.12. `slinput` function.**

---

```
if_sl.c
527 void
528 slinput(c, tp)
529 int     c;
530 struct tty *tp;
531 {
532     struct sl_softc *sc;
533     struct mbuf *m;
534     int      len;
535     int      s;
536     u_char   chdr[CHDR_LEN];

537     tk_nin++;
538     sc = (struct sl_softc *) tp->t_sc;
539     if (sc == NULL)
540         return;
541     if (c & TTY_ERRORMASK || ((tp->t_state & TS_CARR_ON) == 0 &&
542                               (tp->t_cflag & CLOCAL) == 0)) {
543         sc->sc_flags |= SC_ERROR;
544         return;
545     }
546     c &= TTY_CHARMASK;
547     ++sc->sc_if.if_ibytes;
548     switch (c) {
549     case TRANS_FRAME_ESCAPE:
550         if (sc->sc_escape)
551             c = FRAME_ESCAPE;
552         break;
```

```

553     case TRANS_FRAME_END:
554         if (sc->sc_escape)
555             c = FRAME_END;
556         break;
557     case FRAME_ESCAPE:
558         sc->sc_escape = 1;
559         return;
560     case FRAME_END:
561
562         /* FRAME-END code (Figure 5.13) */
563
564     }
565     if (sc->sc_mp < sc->sc_ep) {
566         *sc->sc_mp++ = c;
567         sc->sc_escape = 0;
568         return;
569     }
570     /* can't put lower; would miss an extra frame */
571     sc->sc_flags |= SC_ERROR;
572
573     error:
574     sc->sc_if.if_ierrors++;
575     newpack:
576     sc->sc_mp = sc->sc_buf = sc->sc_ep - SLMAX;
577     sc->sc_escape = 0;
578 }

```

— if\_sl.c

## 527-545

The arguments to `sliinput` are `c`, the next input character; and `tp`, a pointer to the device's `tty` structure. The global integer `tk_nin` counts the incoming characters for all TTY devices. `sliinput` converts `tp->t_sc` to `sc`, a pointer to an `sl_softc` structure. If there is no interface associated with the TTY device, `sliinput` returns immediately.

The first argument to `sliinput` is an integer. In addition to the received character, `c` contains control information sent from the TTY device driver in the high-order bits. If an error is indicated in `c` or the modem-

control lines are not enabled and should not be ignored, SC\_ERROR is set and slinput returns. Later, when slinput processes the END character, the frame is discarded. The CLOCAL flag indicates that the system should treat the line as a local line (i.e., not a dialup line) and should not expect to see modem-control signals.

## 546-636

slinput discards the control bits in c by masking it with TTY\_CHARMASK, updates the count of bytes received on the interface, and jumps based on the received character:

- If c is an escaped ESC character and the *previous* character was an ESC, slinput replaces c with an ESC character.
- If c is an escaped END character and the *previous* character was an ESC, slinput replaces c with an END character.
- If c is the SLIP ESC character,

`sc_escape` is set and `sinput` returns immediately (i.e., the ESC character is discarded).

- If `c` is the SLIP END character, the packet is put on the IP input queue. The processing for the SLIP frame end character is shown in [Figure 5.13](#).

**Figure 5.13. `sinput` function: end-of-frame processing.**

```
560     case FRAME_END:                                     if_sl.c
561         if (sc->sc_flags & SC_ERROR) {
562             sc->sc_flags &= ~SC_ERROR;
563             goto newpack;
564         }
565         len = sc->sc_mp - sc->sc_buf;
566         if (len < 3)
567             /* less than min length packet - ignore */
568             goto newpack;
569         if (sc->sc_bpf) {
570             /*
571              * Save the compressed header, so we
572              * can tack it on later. Note that we
573              * will end up copying garbage in some
574              * cases but this is okay. We remember
575              * where the buffer started so we can
576              * compute the new header length.
577             */
578             bcopy(sc->sc_buf, chdr, CHDR_LEN);
579         }
580         if ((c = (*sc->sc_buf & 0xf0)) != (IPVERSION << 4)) {
581             if (c & 0x80)
582                 c = TYPE_COMPRESSED_TCP;
583             else if (c == TYPE_UNCOMPRESSED_TCP)
584                 *sc->sc_buf &= 0x4f;      /* XXX */
585             /*
586              * We've got something that's not an IP packet.
587              * If compression is enabled, try to decompress it.
588              * Otherwise, if auto-enable compression is on and
589              * it's a reasonable packet, decompress it and then
590              * enable compression. Otherwise, drop it.
591             */
```

```

592         if (sc->sc_if.if_flags & SC_COMPRESS) {
593             len = sl_uncompress_tcp(&sc->sc_buf, len,
594                                     (u_int) c, &sc->sc_comp);
595             if (len <= 0)
596                 goto error;
597         } else if ((sc->sc_if.if_flags & SC_AUTOCOMP) &&
598                     c == TYPE_UNCOMPRESSED_TCP && len >= 40) {
599             len = sl_uncompress_tcp(&sc->sc_buf, len,
600                                     (u_int) c, &sc->sc_comp);
601             if (len <= 0)
602                 goto error;
603             sc->sc_if.if_flags |= SC_COMPRESS;
604         } else
605             goto error;
606     }
607     if (sc->sc_bpf) {
608         /*
609          * Put the SLIP pseudo-"link header" in place.
610          * We couldn't do this any earlier since
611          * decompression probably moved the buffer
612          * pointer. Then, invoke BPF.
613          */
614         u_char *hp = sc->sc_buf - SLIP_HDRLEN;
615
616         hp[SLX_DIR] = SLIPDIR_IN;
617         bcopy(chdr, &hp[SLX_CHDR], CHDR_LEN);
618         bpf_tap(sc->sc_bpf, hp, len + SLIP_HDRLEN);
619
620         m = sl_btom(sc, len);
621         if (m == NULL)
622             goto error;
623
624         sc->sc_if.ipackets++;
625         sc->sc_if.if_lastchange = time;
626         s = splimp();
627         if (IF_QFULL(&ipintrq)) {
628             IF_DROP(&ipintrq);
629             sc->sc_if.if_ierrors++;
630             sc->sc_if.if_iqdrops++;
631             m_freem(m);
632         } else {
633             IF_ENQUEUE(&ipintrq, m);
634             schednetisr(NETISR_IP);
635         }
636         splx(s);
637         goto newpack;

```

---

if\_sl.c

The common flow of control through this switch statement is to fall through (there is no default case). Most bytes are data and don't match any of the four cases. Control also falls through the switch in the first two cases.

637-649

If control falls through the switch, the received character is part of the IP packet. The character is stored in the cluster (if there is room), the pointer is advanced, sc\_escape is cleared, and slinput returns.

If the cluster is full, the character is discarded and slinput sets SC\_ERROR. Control reaches error when the cluster is full or when an error is detected in the end-of-frame processing. At newpack the cluster pointers are reset for a new packet, sc\_escape is cleared, and slinput returns.

[Figure 5.13](#) shows the FRAME\_END code omitted from [Figure 5.12](#).

560-579

slinput discards an incoming SLIP packet immediately if SC\_ERROR was set while the packet was being received or if the packet is less than 3 bytes in length (remember that the packet may be compressed).

If the SLIP interface is tapped by BPF, slinput saves a copy of the (possibly

compressed) header in the chdr array.

## 580-606

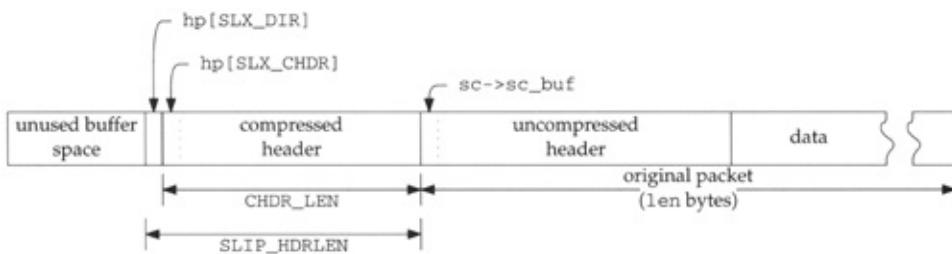
By examining the first byte of the packet, slinput determines if it is an uncompressed IP packet, a compressed TCP segment, or an uncompressed TCP segment. The type is saved in c and the type information is removed from the first byte of data ([Section 29.13](#)). If the packet appears to be compressed and compression is enabled, sl\_uncompress\_tcp attempts to uncompress the packet. If compression is not enabled, auto-enable compression is on, and if the packet is large enough sl\_uncompress\_tcp is also called. If it is a compressed TCP packet, the compression flag is set.

slinput discards packets it does not recognize by jumping to error. [Section 29.13](#) discusses the header compression techniques in more detail. The cluster now contains a complete uncompressed packet.

## 607-618

After SLIP has decompressed the packet, the header and data are passed to BPF. [Figure 5.14](#) shows the layout of the buffer constructed by slinput.

**Figure 5.14. SLIP packet in BPF format.**



The first byte of the BPF header encodes the direction of the packet, in this case incoming (SLIPDIR\_IN). The next 15 bytes contain the compressed header. The entire packet is passed to bpf\_tap.

619-635

sl\_btom converts the cluster to an mbuf chain. If the packet is small enough to fit in a single mbuf, sl\_btom copies the packet from the cluster to a newly allocated mbuf packet header; otherwise sl\_btom attaches the cluster to an mbuf and allocates a new cluster for the

interface. This is faster than copying from one cluster to another. We do not show `sl_btom` in this text.

Since only IP packets are transmitted on a SLIP interface, `sliinput` does not have to select a protocol queue (as it does in the Ethernet driver). The packet is queued on `ipintrq`, an IP software interrupt is scheduled, and `sliinput` jumps to `newpack`, where it updates the cluster packet pointers and clears `sc_escape`.

While the SLIP driver increments `if_ierrors` if the packet cannot be queued on `ipintrq`, neither the Ethernet nor loopback drivers increment this statistic in the same situation.

Access to the IP input queue must be protected by `splimp` even though `sliinput` is called at `spltty`. Recall from [Figure 1.14](#) that an `splimp` interrupt can preempt `spltty` processing.

## SLIP Output Processing: `sloutput`

As with all network interfaces, output processing begins when a network-level protocol calls the interface's `if_output` function. For the Ethernet driver, the function is `ether_output`. For SLIP, the function is `sloutput` ([Figure 5.15](#)).

### **Figure 5.15. `sloutput` function.**

```
259 int
260 sloutput(ifp, m, dst, rtp)
261 struct ifnet *ifp;
262 struct mbuf *m;
263 struct sockaddr *dst;
264 struct rtentry *rtp;
265 {
266     struct sl_softc *sc = &sl_softc[ifp->if_unit];
267     struct ip *ip;
268     struct ifqueue *ifq;
269     int     s;
270     /*
271      * Cannot happen (see slioctl).  Someday we will extend
272      * the line protocol to support other address families.
273      */
274     if (dst->sa_family != AF_INET) {
275         printf("sl%d: af%d not supported\n", sc->sc_if.if_unit,
276                dst->sa_family);
277         m_free(m);
278         sc->sc_if.if_noproto++;
279         return (EAFNOSUPPORT);
280     }
281     if (sc->sc_ttyp == NULL) {
282         m_free(m);
283         return (ENETDOWN);      /* sort of */
284     }
285     if ((sc->sc_ttyp->t_state & TS_CARR_ON) == 0 &&
286         (sc->sc_ttyp->t_cflag & CLOCAL) == 0) {
287         m_free(m);
288         return (EHOSTUNREACH);
289     }
290     ifq = &sc->sc_if.if_snd;
291     ip = mtod(m, struct ip *);
292     if (sc->sc_if.if_flags & SC_NOICMP && ip->ip_p == IPPROTO_ICMP) {
293         m_free(m);
294         return (ENETRESET);    /* XXX ? */
295     }
296     if (ip->ip_tos & IPTOS_LOWDELAY)
297         ifq = &sc->sc_fastq;
298     s = splimp();
299     if (IF_QFULL(ifq)) {
300         IF_DROP(ifq);
301         m_free(m);
302         splx(s);
303         sc->sc_if.if_oerrors++;
304         return (ENOBUFS);
305     }
306     IF_ENQUEUE(ifq, m);
307     sc->sc_if.if_lastchange = time;
308     if (sc->sc_ttyp->t_outq.c_cc == 0)
309         slstart(sc->sc_ttyp);
310     splx(s);
311     return (0);
312 }
```

---

if\_sl.c

259-289

The four arguments to sloutput are: ifp, a pointer to the SLIP ifnet structure (in this

case an sl\_softc structure); m, a pointer to the packet to be queued for output; dst, the next-hop destination for the packet; and rtp, a pointer to a route entry. The fourth argument is not used by sloutput, but it is required since sloutput must match the prototype for the if\_output function in the ifnet structure.

sloutput ensures that dst is an IP address, that the interface is connected to a TTY device, and that the TTY device is operating (i.e., the carrier is on or should be ignored). An error is returned immediately if any of these tests fail.

290-291

The SLIP interface maintains two queues of outgoing packets. The standard queue, if\_snd, is selected by default.

292-295

If the outgoing packet contains an ICMP message and SC\_NOICMP is set for the interface, the packet is discarded. This prevents a SLIP link from being

overwhelmed by extraneous ICMP packets (e.g., ECHO packets) sent by a malicious user ([Chapter 11](#)).

The error code ENETRESET indicates that the packet was discarded because of a policy decision (versus a network failure). We'll see in [Chapter 11](#) that the error is silently discarded unless the ICMP message was generated locally, in which case an error is returned to the process that tried to send the message.

Net/2 returned a 0 in this case. To a diagnostic tool such as ping or traceroute it would appear as if the packet disappeared since the output operation would report a successful completion.

In general, ICMP messages can be discarded. They are not required for correct operation, but discarding them makes troubleshooting more difficult and may lead to less than optimal routing decisions, poorer performance, and wasted network resources.

296-297

If the TOS field in the outgoing packet specifies low-delay service (IPTOS\_LOWDELAY), the output queue is changed to sc\_fastq.

RFC 1700 and RFC 1349 [[Almquist 1992](#)] specify the TOS settings for the standard protocols. Low-delay service is specified for Telnet, Rlogin, FTP (control), TFTP, SMTP (command phase), and DNS (UDP query). See Section 3.2 of Volume 1 for more details.

In previous BSD releases, the ip\_tos was not set correctly by applications. The SLIP driver implemented TOS queueing by examining the transport headers contained within the IP packet. If it found TCP packets for the FTP (command), Telnet, or Rlogin ports, the packet was queued as if IPTOS\_LOWDELAY was specified. Many routers continue this practice, since many implementations of these interactive services still do not set

ip\_tos.

298-312

The packet is now placed on the selected queue, the interface statistics are updated, and (if the TTY output queue is empty) sloutput calls slstart to initiate transmission of the packet.

SLIP increments if\_oerrors if the interface queue is full; ether\_output does not.

Unlike the Ethernet output function (ether\_output), sloutput does not construct a data-link header for the outgoing packet. Since the only other system on a SLIP network is at the other end of the serial link, there is no need for hardware addresses or a protocol, such as ARP, to convert between IP addresses and hardware addresses. Protocol identifiers (such as the Ethernet *type* field) are also superfluous, since a SLIP link carries only IP packets.

## slstart Function

In addition to the call by sloutput, the TTY device driver calls slstart when it drains its output queue and needs more bytes to transmit. The TTY subsystem manages its queues through a clist structure. In [Figure 5.8](#) the output clist t\_outq is shown below slstart and above the device's t\_oproc function. slstart adds bytes to the queue, while t\_oproc drains the queue and transmits the bytes.

The slstart function is shown in [Figure 5.16](#).

### **Figure 5.16. slstart function: packet dequeuing.**

```
318 void
319 slistart(tp)
320 struct tty *tp;
321 {
322     struct sl_softc *sc = (struct sl_softc *) tp->t_sc;
323     struct mbuf *m;
324     u_char *cp;
325     struct ip *ip;
326     int    s;
327     struct mbuf *m2;
328     u_char  bpfbuf[SLMTU + SLIP_HDRLEN];
329     int    len;
330     extern int cfreecount;
331
332     for (;;) {
333         /*
334          * If there is more in the output queue, just send it now.
335          * We are being called in lieu of ttstart and must do what
336          * it would.
337         */
338         if (tp->t_outq.c_cc != 0) {
339             (*tp->t_oproc) (tp);
340             if (tp->t_outq.c_cc > SLIP_HIWAT)
341                 return;
342         /*
343          * This happens briefly when the line shuts down.
344         */
345         if (sc == NULL)
346             return;
347
348         /*
349          * Get a packet and send it to the interface.
350         */
351         s = splimp();
352         IF_DEQUEUE(&sc->sc_fastq, m);
353         if (m)
354             sc->sc_if.if_omcasts++; /* XXX */
355         else
356             IF_DEQUEUE(&sc->sc_if.if_snd, m);
357         splx(s);
358         if (m == NULL)
359             return;
360
361         /*
362          * We do the header compression here rather than in sloutput
363          * because the packets will be out of order if we are using TOS
364          * queueing, and the connection id compression will get
365          * munged when this happens.
366         */
367         if (sc->sc_bpf) {
368             /*
369              * We need to save the TCP/IP header before it's
370              * compressed. To avoid complicated code, we just
371              * copy the entire packet into a stack buffer (since
```

```

370             * this is a serial line, packets should be short
371             * and/or the copy should be negligible cost compared
372             * to the packet transmission time).
373             */
374         struct mbuf *ml = m;
375         u_char *cp = bpfbuf + SLIP_HDRLEN;
376         len = 0;
377         do {
378             int      mlen = ml->m_len;
379             bcopy(mtod(ml, caddr_t), cp, mlen);
380             cp += mlen;
381             len += mlen;
382         } while (ml = ml->m_next);
383     }
384     if ((ip = mtod(m, struct ip *))->ip_p == IPPROTO_TCP) {
385         if (sc->sc_if.if_flags & SC_COMPRESS)
386             *mtod(m, u_char *) |= sl_compress_tcp(m, ip,
387                                         &sc->sc_comp, 1);
388     }
389     if (sc->sc_bpf) {
390         /*
391          * Put the SLIP pseudo-"link header" in place.  The
392          * compressed header is now at the beginning of the
393          * mbuf.
394         */
395         bpfbuf[SLX_DIR] = SLIPDIR_OUT;
396         bcopy(mtod(m, caddr_t), &bpfbuf[SLX_CHDR], CHDR_LEN);
397         bpf_tap(sc->sc_bpf, bpfbuf, len + SLIP_HDRLEN);
398     }

```

/\* packet output code \*/

---

```

483     }
484 }
```

—if\_sl.c

## 318-358

When slstart is called, tp points to the device's tty structure. The body of slstart consists of a single for loop. If the output queue t\_outq is not empty, slstart calls the output function for the device, t\_oproc, which transmits as many bytes as the device will accept. If more than 100 bytes (SLIP\_HIWAT) remain in the TTY output queue, slstart returns instead of adding another packet's worth of bytes to the

queue. The output device generates an interrupt when it has transmitted all the bytes, and the TTY subsystem calls slstart when the output list is empty.

If the TTY output queue is empty, a packet is dequeued from sc\_fastq or, if sc\_fastq is empty, from the if\_snd queue, thus transmitting all interactive packets before any other packets.

There are no standard SNMP variables to count packets queued according to the TOS fields. The XXX comment in line 353 indicates that the SLIP driver is counting low-delay packets in if\_omcasts, *not* multicast packets.

359-383

If the SLIP interface is tapped by BPF, slstart makes a copy of the output packet before any header compression occurs. The copy is saved on the stack in the bpfbuf array.

384-388

If compression is enabled and the packet

contains a TCP segment, sloutput calls `sl_compress_tcp`, which attempts to compress the packet. The resulting packet type is returned and logically ORed with the first byte in IP header ([Section 29.13](#)).

389-398

The compressed header is now copied into the BPF header, and the direction recorded as `SLIPDIR_OUT`. The completed BPF packet is passed to `bpf_tap`.

483-484

`slstart` returns if the for loop terminates.

The next section of `slstart` ([Figure 5.17](#)) discards packets if the system is low on memory, and implements a simple technique for discarding data generated by noise on the serial line. This is the code omitted from [Figure 5.16](#).

**Figure 5.17. `slstart` function: resource shortages and line noise.**

```
399     sc->sc_if.if_lastchange = time;
400     /*
401      * If system is getting low on clists, just flush our
402      * output queue (if the stuff was important, it'll get
403      * retransmitted).
404      */
405     if (cframerelease < CLISTRESERVE + SLMTU) {
406         m_freem(m);
407         sc->sc_if.if_collisions++;
408         continue;
409     }
410     /*
411      * The extra FRAME_END will start up a new packet, and thus
412      * will flush any accumulated garbage. We do this whenever
413      * the line may have been idle for some time.
414      */
415     if (tp->t_outq.c_cc == 0) {
416         ++sc->sc_if.if_obytes;
417         (void) putc(FRAME_END, &tp->t_outq);
418     }
```

—if\_sl.c

## 399-409

If the system is low on clist structures, the packet is discarded and counted as a collision. By continuing the loop instead of returning, slstart quickly discards all remaining packets queued for output. Each iteration discards a packet, since the device still has too many bytes queued for output. Higher-level protocols must detect the lost packets and retransmit them.

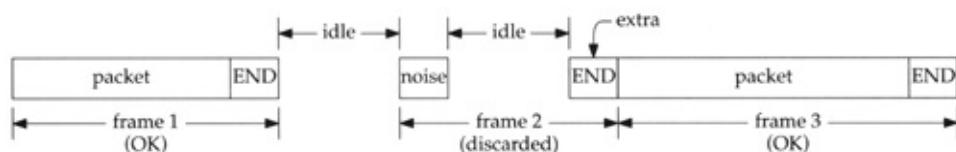
## 410-418

If the TTY output queue is empty, the communication line may have been idle for a period of time and the receiver at the other end may have received extraneous data created by line noise. slstart places

an extra SLIP END character in the output queue. A 0-length frame or a frame created by noise on the line should be discarded by the SLIP interface or IP protocol at the receiver.

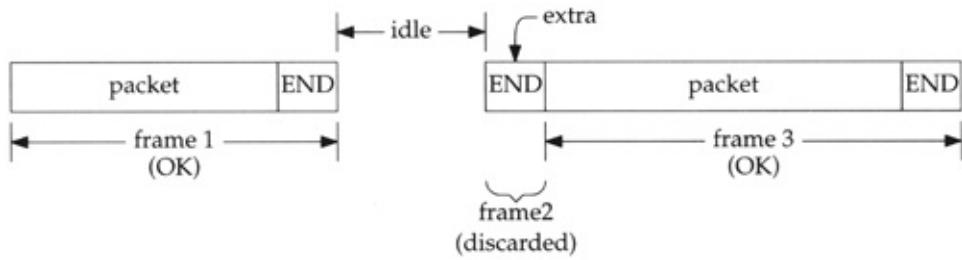
[Figure 5.18](#) illustrates this technique for discarding line noise and is attributed to Phil Karn in RFC 1055. In [Figure 5.18](#), the second end-of-frame (END) is transmitted because the line was idle for a period of time. The invalid frame created by the noise and the END byte is discarded by the receiving system.

**Figure 5.18. Karn's method for discarding noise on a SLIP line.**



In [Figure 5.19](#) there is no noise on the line and the 0-length frame is discarded by the receiving system.

**Figure 5.19. Karn's method with no noise.**



The next section of s1start (Figure 5.20) transfers the data from an mbuf to the output queue for the TTY device.

**Figure 5.20. s1start function: packet transmission.**

```

419     while (m) {
420         u_char *ep;
421         cp = mtod(m, u_char *);
422         ep = cp + m->m_len;
423         while (cp < ep) {
424             /*
425              * Find out how many bytes in the string we can
426              * handle without doing something special.
427              */
428             u_char *bp = cp;
429             while (cp < ep) {
430                 switch (*cp++) {
431                     case FRAME_ESCAPE:
432                     case FRAME_END:
433                         --cp;
434                         goto out;
435                 }
436             }
437             out:
438             if (cp > bp) {
439                 /*
440                  * Put n characters at once
441                  * into the tty output queue.
442                  */
443                 if (b_to_q((char *) bp, cp - bp,
444                           &tp->t_outq))
445                     break;
446                 sc->sc_if.if_obytes += cp - bp;
447             }
448             /*
449              * If there are characters left in the mbuf,
450              * the first one must be special..
451              * Put it out in a different form.
452              */
453             if (cp < ep) {
454                 if (putc(FRAME_ESCAPE, &tp->t_outq))
455                     break;
456                 if (*cp++ == FRAME_ESCAPE ?
457                     TRANS_FRAME_ESCAPE : TRANS_FRAME_END,
458                     &tp->t_outq)) {
459                     (void) unputc(&tp->t_outq);
460                     break;
461                 }
462                 sc->sc_if.if_obytes += 2;
463             }
464         }
465         MFREE(m, m2);
466         m = m2;
467     }

```

---

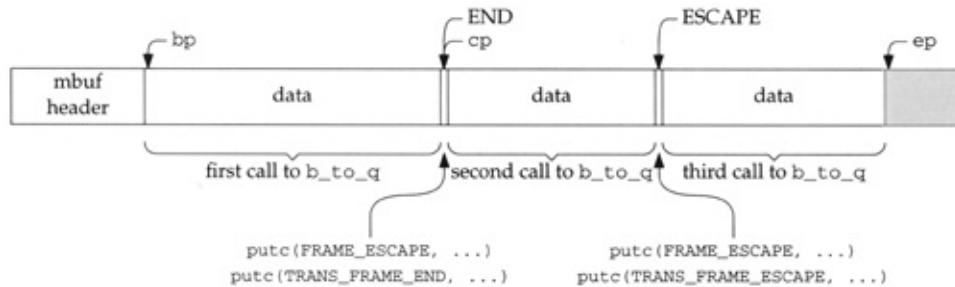
if\_sl.c

## 419-467

The outer while loop in this section is executed once for each mbuf in the chain. The middle while loop transfers the data from each mbuf to the output device. The

inner while loop advances cp until it finds an END or ESC character. b\_to\_q transfers the bytes between bp and cp. END and ESC characters are escaped and queued with two calls to putc. This middle loop is repeated until all the bytes in the mbuf are passed to the TTY device's output queue. [Figure 5.21](#) illustrates this process with an mbuf containing a SLIP END character and a SLIP ESC character.

## Figure 5.21. SLIP transmission of a single mbuf.



bp marks the beginning of the first section of the mbuf to transfer with b\_to\_q, and cp marks the end of the first section. ep marks the end of the data in the mbuf.

If b\_to\_q or putc fail (i.e., data cannot be queued on the TTY device), the break

causes slstart to fall out of the middle while loop. The failure indicates that the kernel has run out of clist resources. After each mbuf is copied to the TTY device, or when an error occurs, the mbuf is released, m is advanced to the next mbuf in the chain, and the outer while loop continues until all the mbufs in the chain have been processed.

[Figure 5.22](#) shows the processing done by slstart to complete the outgoing frame.

## Figure 5.22. slstart function: end-of-frame processing.

```
if_sl.c
468     if (putc(FRAME_END, &tp->t_outq)) {
469         /*
470          * Not enough room. Remove a char to make room
471          * and end the packet normally.
472          * If you get many collisions (more than one or two
473          * a day) you probably do not have enough clists
474          * and you should increase "nclist" in param.c.
475          */
476         (void) unputc(&tp->t_outq);
477         (void) putc(FRAME_END, &tp->t_outq);
478         sc->sc_if.if_collisions++;
479     } else {
480         +*sc->sc_if.if_obytes;
481         sc->sc_if.if_opackets++;
482     }
if_sl.c
```

468-482

Control reaches this code when the outer

while loop has finished queueing the bytes on the output queue. The driver sends a SLIP END character, which terminates the frame.

If an error occurred while queueing the bytes, the outgoing frame is invalid and is detected by the receiving system because of an invalid checksum or length.

Whether or not the frame is terminated because of an error, if the END character does not fit on the output queue, the *last* character on the queue is discarded and slstart ends the frame. This guarantees that an END character is transmitted. The invalid frame is discarded at the destination.

## SLIP Packet Loss

The SLIP interface provides a good example of a best-effort service. SLIP discards packets if the TTY is overloaded; it truncates packets if resources are unavailable after the packet transmission has started, and it inserts extraneous null

packets to detect and discard line noise. In each of these cases, no error message is generated. SLIP depends on IP and the transport layers to detect damaged and missing packets.

On a router forwarding packets from a fast interface such as Ethernet to a low-speed SLIP line, a large percentage of packets are discarded if the sender does not recognize the bottleneck and respond by throttling back the data rate. In [Section 25.11](#) we'll see how TCP detects and responds to this condition. Applications using a protocol without flow control, such as UDP, must recognize and respond to this condition on their own ([Exercise 5.8](#)).

## SLIP Performance Considerations

The MTU of a SLIP frame (SLMTU), the clist high-water mark (SLIP\_HIWAT), and SLIP's TOS queueing strategies are all designed to minimize the delay inherent in a slow serial link for interactive traffic.

### **1. A small MTU improves the delay for**

**interactive data (such as keystrokes and echoes), but hurts the throughput for bulk data transfer. A large MTU improves bulk data throughput, but increases interactive delays. Another problem with SLIP links is that a single typed character is burdened with 40 bytes of TCP and IP header information, which increases the communication delay.**

**The solution is to pick an MTU large enough to provide good interactive response time and decent bulk data throughput, and to compress TCP/IP headers to reduce the per-packet overhead. RFC 1144 [Jacobson 1990a] describes a compression scheme and the timing calculations that result in selecting an MTU of 296 for a typical 9600 bits/sec asynchronous SLIP link. We describe Compressed SLIP (CSLIP) in Section 29.13. Sections 2.10 and 7.2 of Volume 1 summarize the timing considerations and illustrate the**

## **delay on SLIP links.**

- If too many bytes are buffered in the clist (because SLIP\_HIWAT is set too high), the TOS queueing will be thwarted as new interactive traffic waits behind the large amount of buffered data. If SLIP passes 1 byte at a time to the TTY driver (because SLIP\_HIWAT is set too low), the device calls slstart for each byte and the line is idle for a brief period of time after each byte is transferred. Setting SLIP\_HIWAT to 100 minimizes the amount of data queued at the device and reduces the frequency at which the TTY subsystem must call slstart to approximately once every 100 characters.
- As described, the SLIP driver provides TOS queueing by transmitting interactive traffic from the sc\_fastq queue before other traffic on the standard interface queue, if\_snd.

## **slclose Function**

For completeness, we show the slclose

function, which is called when the slattach program closes SLIP's TTY device and terminates the connection to the remote system.

### Figure 5.23. slclose function.

```
-----if_sl.c
210 void
211 slclose(tp)
212 struct tty *tp;
213 {
214     struct sl_softc *sc;
215     int     s;
216     ttywflush(tp);
217     s = splimp();           /* actually, max(spltty, splnet) */
218     tp->t_line = 0;
219     sc = (struct sl_softc *) tp->t_sc;
220     if (sc != NULL) {
221         if_down(&sc->sc_if);
222         sc->sc_ttyp = NULL;
223         tp->t_sc = NULL;
224         MCLFREE((caddr_t) (sc->sc_ep - SLBUFSIZE));
225         sc->sc_ep = 0;
226         sc->sc_mp = 0;
227         sc->sc_buf = 0;
228     }
229     splx(s);
230 }
-----if_sl.c
```

210-230

tp points to the TTY device to be closed. slclose flushes any remaining data out to the serial device, blocks TTY and network processing, and resets the TTY to the default line discipline. If the TTY device is attached to a SLIP interface, the interface is shut down, the links between the two

structures are severed, the mbuf cluster associated with the interface is released, and the pointers into the now-discarded cluster are reset. Finally, splx reenables the TTY and network interrupts.

## slioctl Function

Recall that a SLIP interface has two roles to play in the kernel:

- as a network interface, and
- as a TTY line discipline.

[Figure 5.7](#) indicated that slioctl processes ioctl commands issued for a SLIP interface through a socket descriptor. In [Section 4.4](#) we showed how ifioctl calls slioctl. We'll see a similar pattern for ioctl commands that we cover in later chapters.

[Figure 5.7](#) also indicated that slioctl processes ioctl commands issued for the TTY device associated with a SLIP network interface. The one command recognized by slioctl is shown in [Figure 5.24](#).

## Figure 5.24. slioctl commands.

Command	Argument	Function	Description
SLIOCGUNIT	int *	slioctl	return interface unit associated with the TTY device

The slioctl function is shown in Figure 5.25.

## Figure 5.25. slioctl function.

```
if_sl.c
236 int
237 slioctl(tp, cmd, data, flag)
238 struct tty *tp;
239 int cmd;
240 caddr_t data;
241 int flag;
242 {
243     struct sl_softc *sc = (struct sl_softc *) tp->t_sc;
244     switch (cmd) {
245         case SLIOCGUNIT:
246             *(int *) data = sc->sc_if.if_unit;
247             break;
248         default:
249             return (-1);
250     }
251     return (0);
252 }
```

if\_sl.c

236-252

The t\_sc pointer in the tty structure points to the associated sl\_softc structure. The unit number of the SLIP interface is copied from if\_unit to \*data, which is eventually returned to the process (Section 17.5).

`if_unit` is initialized by `slattach` when the system is initialized, and `t_sc` is initialized by `slopen` when the `slattach` program selects the SLIP line discipline for the TTY device. Since the mapping between a TTY device and a SLIP `sl_softc` structure is established at run time, a process can discover the interface structure selected by the `SLIOCGUNIT` command.

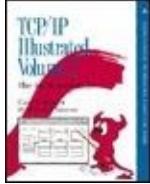
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



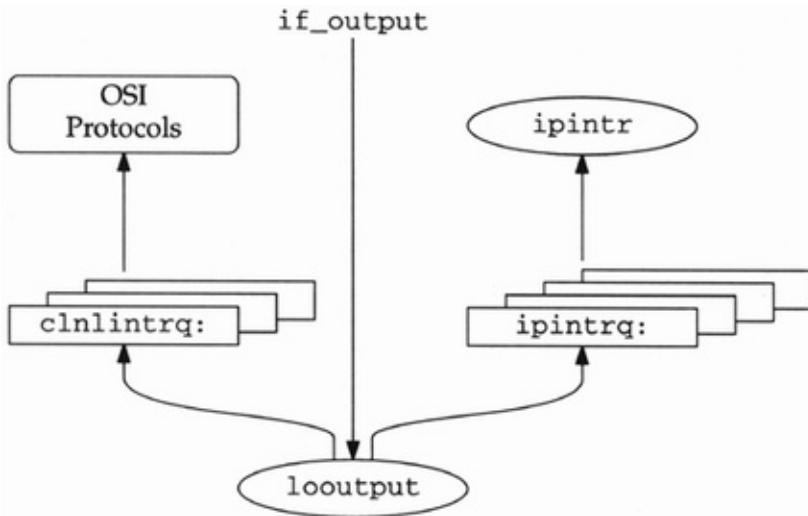
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 5. Interfaces: SLIP and Loopback

### 5.4 Loopback Interface

Any packets sent to the loopback interface ([Figure 5.26](#)) are immediately queued for input. The interface is implemented entirely in software.

**Figure 5.26. Loopback device driver.**



`looutput`, the `if_output` function for the loopback interface, places outgoing packets on the input queue for the protocol specified by the packet's destination address.

We already saw that `ether_output` may call `looutput` to queue a copy of an outgoing broadcast packet when the device has set `IFF_SIMPLEX`. In [Chapter 12](#), we'll see that multicast packets may be also be looped back in this way. `looutput` is shown in [Figure 5.27](#).

**Figure 5.27. The `looutput` function.**

```
57 int
58 looutput(ifp, m, dst, rt)
59 struct ifnet *ifp;
60 struct mbuf *m;
61 struct sockaddr *dst;
62 struct rtentry *rt;
63 {
64     int      s, isr;
65     struct ifqueue *ifq = 0;
66     if ((m->m_flags & M_PKTHDR) == 0)
67         panic("looutput no HDR");
68     ifp->if_lastchange = time;
69     if (loif.if_bpf) {
70         /*
71          * We need to prepend the address family as
72          * a four byte field.  Cons up a dummy header
```

```

73         * to pacify bpf. This is safe because bpf
74         * will only read from the mbuf (i.e., it won't
75         * try to free it or keep a pointer to it).
76         */
77     struct mbuf m0;
78     u_int    af = dst->sa_family;
79
80     m0.m_next = m;
81     m0.m_len = 4;
82     m0.m_data = (char *) &af;
83
84     bpf_mtap(loif.if_bpf, &m0);
85 }
86 m->m_pkthdr.rcvif = ifp;
87
88 if (rt && rt->rt_flags & (RTF_REJECT | RTF_BLACKHOLE)) {
89     m_freem(m);
90     return (rt->rt_flags & RTF_BLACKHOLE ? 0 :
91             rt->rt_flags & RTF_HOST ? EHOSTUNREACH : ENETUNREACH);
92 }
93 ifp->if_opackets++;
94 ifp->if_oBYTES += m->m_pkthdr.len;
95 switch (dst->sa_family) {
96 case AF_INET:
97     ifq = &ipintrq;
98     isr = NETISR_IP;
99     break;
100
101 case AF_ISO:
102     ifq = &clnlintrq;
103     isr = NETISR_ISO;
104     break;
105
106 default:
107     printf("lo%d: can't handle af%d\n", ifp->if_unit,
108           dst->sa_family);
109     m_freem(m);
110     return (EAFNOSUPPORT);
111 }
112 s = splimp();
113 if (IF_QFULL(ifq)) {
114     IF_DROP(ifq);
115     m_freem(m);
116     splx(s);
117     return (ENOBUFS);
118 }
119 IF_ENQUEUE(ifq, m);
120 schednetisr(isr);
121 ifp->if_ipackets++;
122 ifp->if_iBYTES += m->m_pkthdr.len;
123 splx(s);
124 return (0);
125 }

```

if\_loop.c

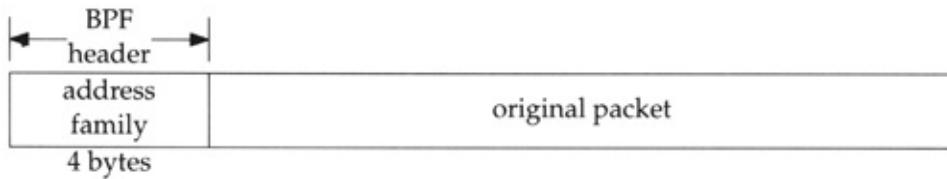
## 57-68

The arguments to looutput are the same as those to ether\_output since both are called indirectly through the if\_output pointer in their ifnet structures: ifp, a

pointer to the outgoing interface's ifnet structure; m, the packet to send; dst, the destination address of the packet; and rt, routing information. If the first mbuf on the chain does not contain a packet, looutput calls panic.

[Figure 5.28](#) shows the logical layout for a BPF loopback packet.

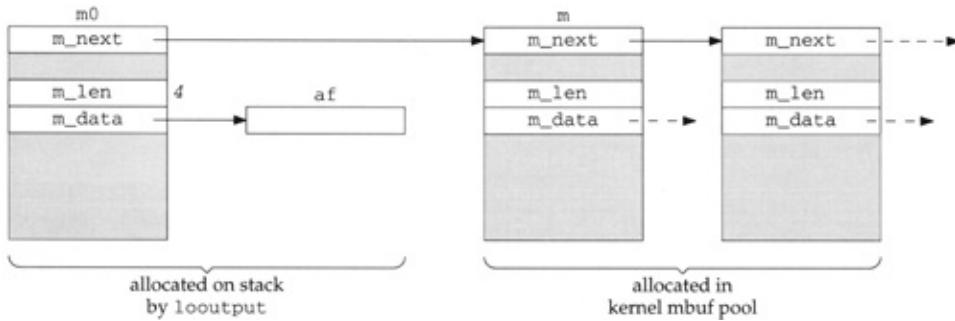
### Figure 5.28. BPF loopback packet: logical format.



69-83

The driver constructs the BPF loopback packet header in m0 on the stack and connects m0 to the mbuf chain containing the original packet. Note the unusual declaration of m0. It is an *mbuf*, not a pointer to an mbuf. m\_data in m0 points to af, which is also allocated on the stack. [Figure 5.29](#) shows this arrangement.

**Figure 5.29. BPF loopback packet: mbuf format.**



`looutput` copies the destination's address family into `af` and passes the new mbuf chain to `bpf_mtap`, which processes the packet. Contrast this to `bpf_tap`, which accepts the packet in a single contiguous buffer not in an mbuf chain. As the comment indicates, BPF never releases mbufs in a chain, so it is safe to pass `m0` (which points to an mbuf on the stack) to `bpf_mtap`.

84-89

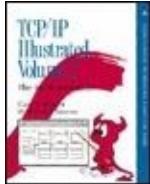
The remainder of `looutput` contains *input* processing for the packet. Even though this is an output function, the packet is being looped back to appear as input. First, `m->m_pkthdr.rcvif` is set to point to

the receiving interface. If the caller provided a routing entry, looutput checks to see if it indicates that the packet should be rejected (RTF\_REJECT) or silently discarded (RTF\_BLACKHOLE). A black hole is implemented by discarding the mbuf and returning 0. It appears to the caller as if the packet has been transmitted. To reject a packet, looutput returns EHOSTUNREACH if the route is for a host and ENETUNREACH if the route is for a network.

The various RTF\_xxx flags are described in [Figure 18.25](#).

## 90-120

looutput then selects the appropriate protocol input queue and software interrupt by examining sa\_family in the packet's destination address. It then queues recognized packets and schedules a software interrupt with schednetisr.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 5. Interfaces: SLIP and Loopback

### 5.5 Summary

We described the two remaining interfaces to which we refer throughout the text: sl0, a SLIP interface, and lo0, the standard loopback interface.

We showed the relationship between the SLIP interface and the SLIP line discipline, described the SLIP encapsulation method, and discussed TOS processing for interactive traffic and other performance considerations for the SLIP driver.

We showed how the loopback interface demultiplexes outgoing packets based on their destination address family and places

the packet on the appropriate input queue.

## Exercises

**5.1** Why does the loopback interface not have an input function?

**5.2** Why do you think `mo` is allocated on the stack in [Figure 5.27](#)?

**5.3** Perform an analysis of SLIP characteristics for a 19,200 bps serial line. Should the SLIP MTU be changed for this line?

**5.4** Derive a formula to select a SLIP MTU based on the speed of the serial line.

**5.5** What happens if a packet is too large to fit in SLIP'S input buffer?

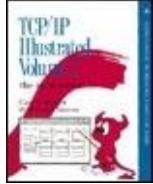
An earlier version of `sinput` did not set

**5.6** SC\_ERROR when a packet overflowed the input buffer. How would the error be detected in this case?

**5.7** In Figure 4.31 le is initialized by indexing the le\_softc array with ifp->if\_unit. Can you think of another method for initializing le?

**5.8** How can a UDP application recognize when its packets are being discarded because of a bottleneck in the network?





[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Chapter 6. IP Addressing

[Section 6.1. Introduction](#)

[Section 6.2. Code Introduction](#)

[Section 6.3. Interface and Address Summary](#)

[Section 6.4. sockaddr\\_in Structure](#)

[Section 6.5. in\\_ifaddr Structure](#)

[Section 6.6. Address Assignment](#)

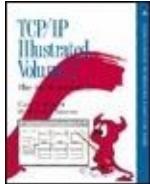
[Section 6.7. Interface ioctl Processing](#)

[Section 6.8. Internet Utility Functions](#)

[Section 6.9. ifnet Utility Functions](#)

[Section 6.10. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

### 6.1 Introduction

This chapter describes how Net/3 manages IP addressing information. We start with the `in_ifaddr` and `sockaddr_in` structures, which are based on the generic `ifaddr` and `sockaddr` structures.

The remainder of the chapter covers IP address assignment and several utility functions that search the interface data structures and manipulate IP addresses.

### IP Addresses

Although we assume that readers are familiar with the basic Internet addressing system, several issues are worth pointing

out.

In the IP model, it is the network interfaces on a system (a host or a router) that are assigned addresses, not the system itself. In the case of a system with multiple interfaces, the system is *multihomed* and has more than one IP address. A router is, by definition, multihomed. As we'll see, this architectural feature has several subtle ramifications.

Five classes of IP addresses are defined. Class A, B, and C addresses support *unicast* communication. Class D addresses support IP *multicasting*. In a multicast communication, a single source sends a datagram to multiple destinations. Class D addresses and multicasting protocols are described in [Chapter 12](#). Class E addresses are experimental. Packets received with class E addresses are discarded by hosts that aren't participating in the experiment.

It is important that we emphasize the difference between *IP multicasting* and *hardware multicasting*. Hardware multicasting is a feature of the data-link

hardware used to transmit packets to multiple hardware interfaces. Some network hardware, such as Ethernet, supports data-link multicasting. Other hardware may not.

IP multicasting is a software feature implemented in IP systems to transmit packets to multiple IP addresses that may be located throughout the internet.

We assume that the reader is familiar with subnetting of IP networks (RFC 950 [[Mogul and Postel 1985](#)] and Chapter 3 of Volume 1). We'll see that each network interface has an associated subnet mask, which is critical in determining if a packet has reached its final destination or if it needs to be forwarded. In general, when we refer to the network portion of an IP address we are including any subnet that may be defined. When we need to differentiate between the network and the subnet, we do so explicitly.

The loopback network, 127.0.0.0, is a special class A network. Addresses of this form must never appear outside of a host.

Packets sent to this network are looped back and received by the host.

RFC 1122 requires that all addresses within the loopback network be handled correctly. Since the loopback interface must be assigned an address, many systems select 127.0.0.1 as the loopback address. If the system is not configured correctly, addresses such as 127.0.0.2 may not be routed to the loopback interface but instead may be transmitted on an attached network, which is prohibited. Some systems may correctly route the packet to the loopback interface where it is dropped since the destination address does not match the configured address: 127.0.0.1.

[Figure 18.2](#) shows a Net/3 system configured to reject packets sent to a loopback address other than 127.0.0.1.

## Typographical Conventions for IP Addresses

We usually display IP addresses in *dotted-decimal* notation. [Figure 6.1](#) lists the range of IP address for each address class.

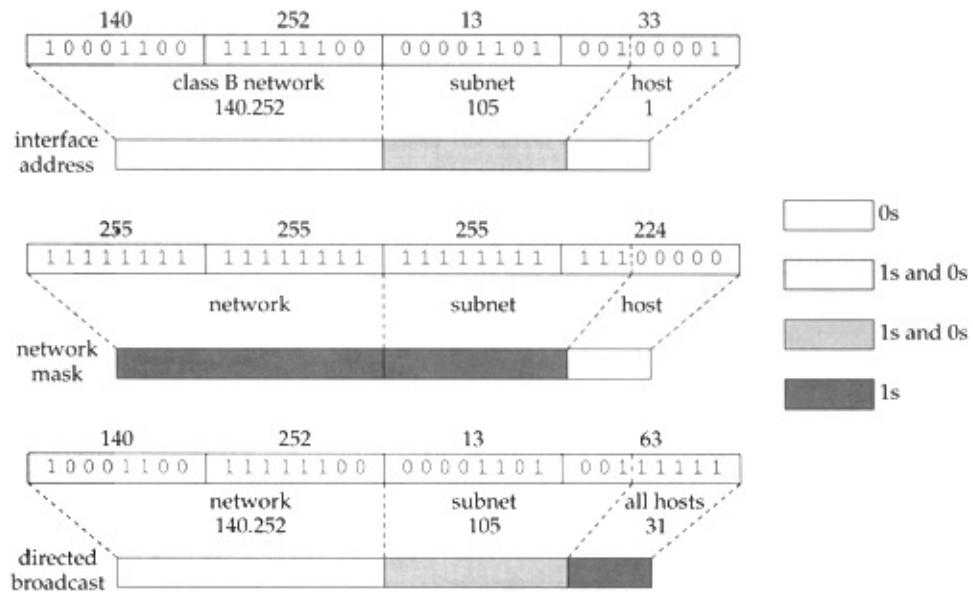
### **Figure 6.1. Ranges for different classes of IP addresses.**

Class	Range	Type
A	0.0.0 to 127.255.255.255	
B	128.0.0.0 to 191.255.255.255	unicast
C	192.0.0.0 to 223.255.255.255	
D	224.0.0.0 to 239.255.255.255	multicast
E	240.0.0.0 to 247.255.255.255	experimental

For some of our examples, the subnet field is not aligned with a byte boundary (i.e., a network/subnet/host division of 16/11/5 in a class B network). It can be difficult to identify the portions of such address from the dotted-decimal notation so we'll also use block diagrams to illustrate the contents of IP addresses. We'll show each address with three parts: network, subnet, and host. The shading of each part indicates its contents. [Figure 6.2](#) illustrates both the block notation and the dotted-decimal notation using the Ethernet interface of the host sun from our sample

network (Section 1.14).

**Figure 6.2. Alternate IP address notations.**



When a portion of the address is not all 0s or all 1s, we use the two intermediate shades. We have two types of intermediate shades so we can distinguish network and subnet portions or to show combinations of address as in Figure 6.31.

## Hosts and Routers

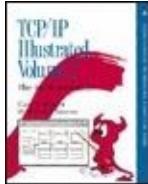
Systems on an internet can generally be divided into two types: *hosts* and *routers*. A host usually has a single network interface and is either the source or destination for an IP packet. A router has multiple network interfaces and forwards packets from one network to the next as the packet moves toward its destination. To perform this function, routers exchange information about the network topology using a variety of specialized routing protocols. IP routing issues are complex, and they are discussed starting in [Chapter 18](#).

A system with multiple network interfaces is still called a *host* if it does not route packets between its network interfaces. A system may be both a host and a router. This is often the case when a router provides transport-level services such as Telnet access for configuration, or SNMP for network management. When the distinction between a host and router is unimportant, we use the term *system*.

Careless configuration of a router can disrupt the normal operation of a network,

so RFC 1122 states that a system must default to operate as a host and must be explicitly configured by an administrator to operate as a router. This purposely discourages administrators from operating general-purpose host computers as routers without careful consideration. In Net/3, a system acts as a router if the global integer ipforwarding is nonzero and as a host if ipforwarding is 0 (the default).

A router is often called a *gateway* in Net/3, although the term *gateway* is now more often associated with a system that provides application-level routing, such as an electronic mail gateway, and not one that forwards IP packets. We use the term *router* and assume that ipforwarding is nonzero in this book. We have also included all code conditionally included when GATEWAY is defined during compilation of the Net/3 kernel, which defines ipforwarding to be 1.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

## 6.2 Code Introduction

The two headers and two C files listed in [Figure 6.3](#) contain the structure definitions and utility functions described in this chapter.

**Figure 6.3. Files discussed in this chapter.**

File	Description
netinet/in.h	Internet address definitions
netinet/in_var.h	Internet interface definitions
netinet/in.c	Internet initialization and utility functions
netinet/if.c	Internet interface utility functions

## Global Variables

The two global variables introduced in this chapter are listed in [Figure 6.4](#).

## Figure 6.4. Global variables introduced in this chapter.

Variable	Datatype	Description
in_ifaddr	struct in_ifaddr *	head of in_ifaddr structure list
in_interfaces	int	number of IP capable interfaces

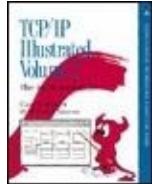
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



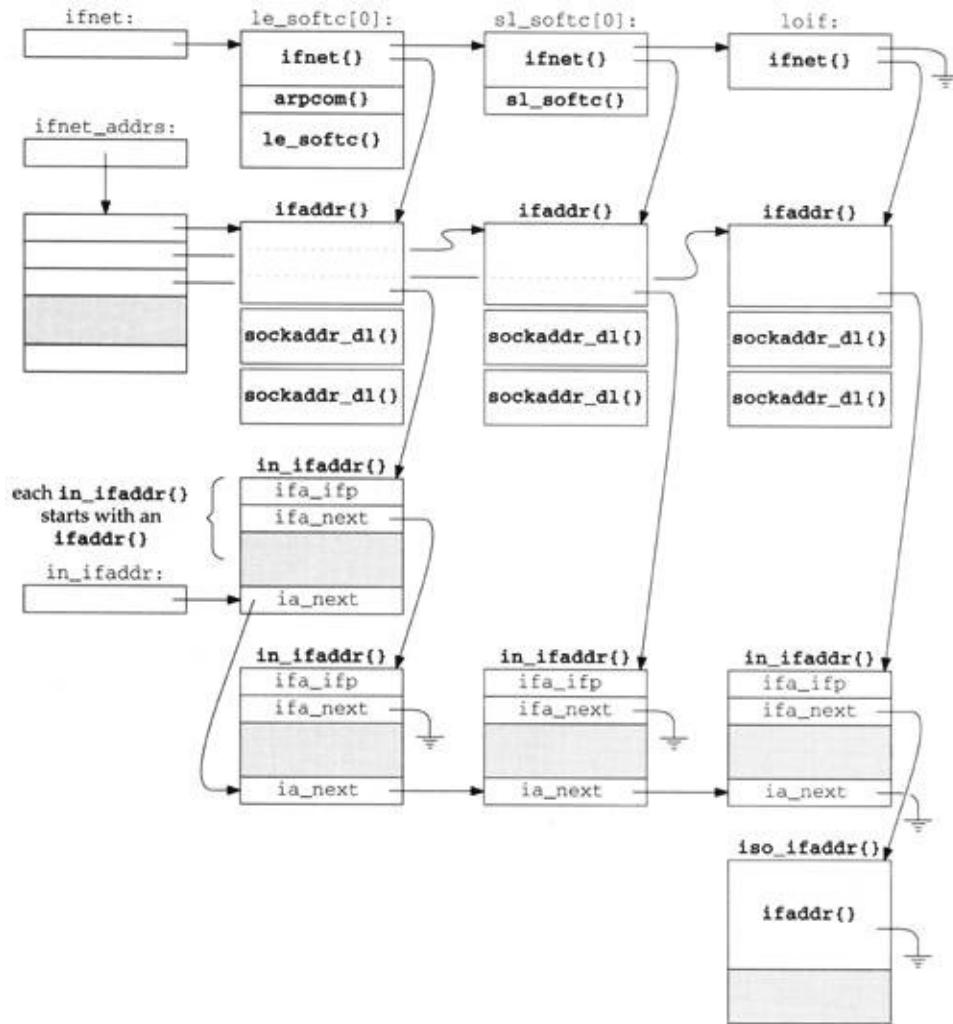
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

### 6.3 Interface and Address Summary

A sample configuration of all the interface and address structures described in this chapter is illustrated in [Figure 6.5](#).

**Figure 6.5. Interface and address data structures.**

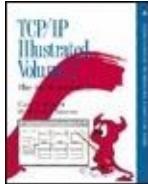


**Figure 6.5** shows our three example interfaces: the Ethernet interface, the SLIP interface, and the loopback interface. All have a link-level address as the first node in their address list. The Ethernet interface is shown with two IP addresses, the SLIP interface with one IP address, and the loopback interface has an IP address and an OSI address.

Note that all the IP addresses are linked into the `in_ifaddr` list and all the link-level addresses can be accessed from the `ifnet_addrs` array.

The `ifa_ifp` pointers within each `ifaddr` structure have been omitted from [Figure 6.5](#) for clarity. The pointers refer back to the `ifnet` structure that heads the list containing the `ifaddr` structure.

The following sections describe the data structures contained in [Figure 6.5](#) and the IP-specific ioctl commands that examine and modify the structures.



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

### 6.4 sockaddr\_in Structure

We discussed the generic sockaddr and ifaddr structures in [Chapter 3](#). Now we show the structures specialized for IP: sockaddr\_in and in\_ifaddr. Addresses in the Internet domain are held in a sockaddr\_in structure:

**Figure 6.6. sockaddr\_in structure.**

```
in.h
68 struct in_addr {
69     u_long s_addr;           /* 32-bit IP address, net byte order */
70 };
71
106 struct sockaddr_in {
107     u_char sin_len;          /* sizeof (struct sockaddr_in) = 16 */
108     u_char sin_family;        /* AF_INET */
109     u_short sin_port;         /* 16-bit port number, net byte order */
110     struct in_addr sin_addr;
111     char sin_zero[8];         /* unused */
112 };
in.h
```

68-70

Net/3 stores 32-bit Internet addresses in network byte order in an `in_addr` structure for historical reasons. The structure has a single member, `s_addr`, which contains the address. That organization is kept in Net/3 even though it is superfluous and clutters the code.

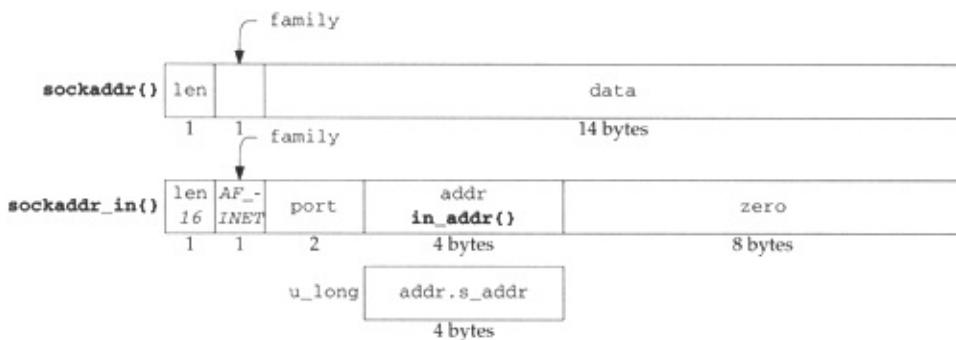
106-112

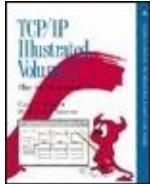
`sin_len` is always 16 (the size of the `sockaddr_in` structure) and `sin_family` is `AF_INET`. `sin_port` is a 16-bit value in network (not host) byte order used to demultiplex transport-level messages. `sin_addr` specifies a 32-bit Internet address.

[Figure 6.6](#) shows that the `sin_port`, `sin_addr`, and `sin_zero` members of `sockaddr_in` overlay the `sa_data` member of `sockaddr`. `sin_zero` is unused in the Internet domain but must consist of all 0 bytes ([Section 22.7](#)). It pads the `sockaddr_in` structure to the length of a `sockaddr` structure.

Usually, when an Internet address is stored in a u\_long it is in host byte order to facilitate comparisons and bit operations on the address. s\_addr within the in\_addr structure (Figure 6.7) is a notable exception.

**Figure 6.7. The organization of a sockaddr\_in structure (sin\_omitted).**





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

### 6.5 in\_ifaddr Structure

Figure 6.8 shows the interface address structure defined for the Internet protocols. For each IP address assigned to an interface, an `in_ifaddr` structure is allocated and added to the interface address list and to the global list of IP addresses (Figure 6.5).

**Figure 6.8. The `in_ifaddr` structure.**

```

41 struct in_ifaddr {
42     struct ifaddr ia_ifa;           /* protocol-independent info */
43 #define ia_ifp      ia_ifa.ifa_ifp
44 #define ia_flags    ia_ifa.ifa_flags
45     struct in_ifaddr *ia_next;    /* next internet addresses list */
46     u_long ia_net;              /* network number of interface */
47     u_long ia_netmask;          /* mask of net part */
48     u_long ia_subnet;           /* subnet number, including net */
49     u_long ia_subnetmask;       /* mask of subnet part */
50     struct in_addr ia_netbroadcast; /* to recognize net broadcasts */
51     struct sockaddr_in ia_addr;  /* space for interface name */
52     struct sockaddr_in ia_dstaddr; /* space for broadcast addr */
53 #define ia_broadaddr ia_dstaddr
54     struct sockaddr_in ia_sockmask; /* space for general netmask */
55     struct in_multi *ia_multiaddrs; /* list of multicast addresses */
56 };

```

in\_var.h

## 41-45

in\_ifaddr starts with the generic interface address structure, ia\_ifa, followed by the IP-specific members. The ifaddr structure was shown in [Figure 3.15](#). The two macros, ia\_ifp and ia\_flags, simplify access to the interface pointer and interface address flags stored in the generic ifaddr structure. ia\_next maintains a linked list of all Internet addresses that have been assigned to any interface. This list is independent of the list of link-level ifaddr structures associated with each interface and is accessed through the global list in\_ifaddr.

## 46-54

The remaining members (other than

`ia_multiaddrs`) are included in [Figure 6.9](#), which shows the values for the three interfaces on sun from our example class B network. The addresses stored as `u_long` variables are kept in host byte order; the `in_addr` and `sockaddr_in` variables are in network byte order. `sun` has a PPP interface, but the information shown in this table is the same for a PPP interface or for a SLIP interface.

**Figure 6.9. Ethernet, PPP, and loopback `in_ifaddr` structures on sun.**

Variable	Type	Ethernet	PPP	Loopback	Description
<code>ia_addr</code>	<code>sockaddr_in</code>				network, subnet, and host numbers
<code>ia_net</code>	<code>u_long</code>				network number
<code>ia_netmask</code>	<code>u_long</code>				network number mask
<code>ia_subnet</code>	<code>u_long</code>				network and subnet number
<code>ia_subnetmask</code>	<code>u_long</code>				network and subnet mask
<code>ia_netbroadcast</code>	<code>in_addr</code>				network broadcast address
<code>ia_broadaddr</code>	<code>sockaddr_in</code>				directed broadcast address
<code>ia_dstaddr</code>	<code>sockaddr_in</code>				destination address
<code>ia_sockmask</code>	<code>sockaddr_in</code>				like <code>ia_subnetmask</code> but in network byte order

The last member of the `in_ifaddr` structure points to a list of `in_multi` structures ([Section 12.6](#)), each of which contains an IP multicast address associated with the interface.

---

## Chapter 6. IP Addressing

---

### 6.6 Address Assignment

In Chapter 4 we showed the initialization of the system at system initialization time. Before the Internet protocol can be used, each interface must be assigned an IP address. Once the Net/Work interface configuration program, which issues configuration commands, has been run, the configuration is normally done by the /etc/netstart shell script,

Figure 6.10 shows the ioctl commands discussed in Chapter 4. Note that the ioctl commands must be from the same address family as the socket on which they are issued (i.e., you can't configure an OSI address on a TCP socket or an ICMP socket). The commands are issued on a UDP socket.

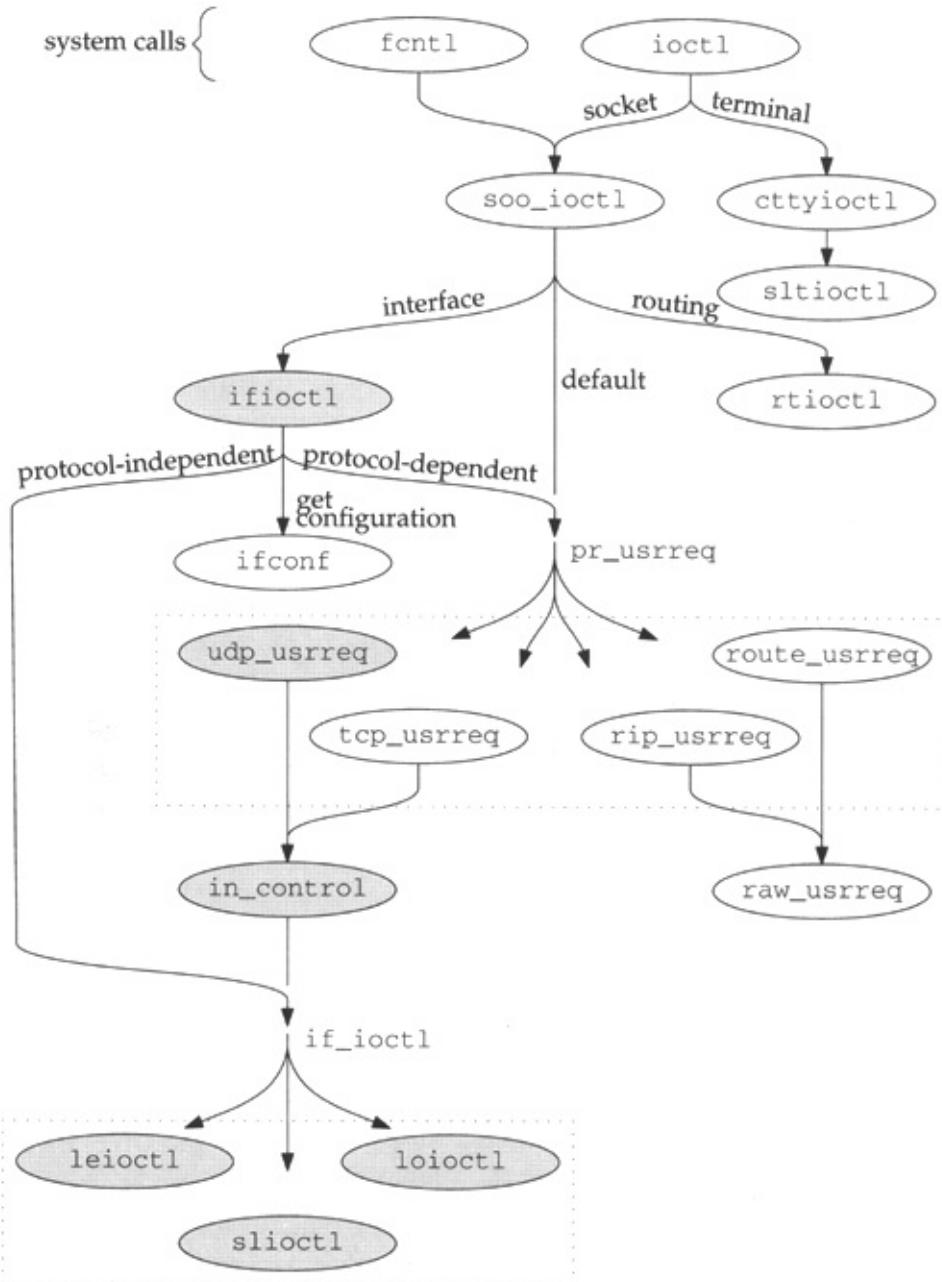
**Figure 6.10. Initialization of a Network Interface**

Command	Argument	Function	Description
<i>SIOCGIFADDR</i>	struct ifreq *	in_control	get interface address
<i>SIOCGIFNETMASK</i>	struct ifreq *	in_control	get interface netmask
<i>SIOCGIFDSTADDR</i>	struct ifreq *	in_control	get interface destination address
<i>SIOCGIFBRDADDR</i>	struct ifreq *	in_control	get interface broadcast address
<i>SIOCSIFADDR</i>	struct ifreq *	in_control	set interface address
<i>SIOCSIFNETMASK</i>	struct ifreq *	in_control	set interface netmask
<i>SIOCSIFDSTADDR</i>	struct ifreq *	in_control	set interface destination address
<i>SIOCSIFBRDADDR</i>	struct ifreq *	in_control	set interface broadcast address
<i>SIOCDIFADDR</i>	struct ifreq *	in_control	delete interface address
<i>SIOCAIFADDR</i>	struct in_aliasreq *	in_control	add interface address

The commands that get address information start with *SIOCG*. The commands that modify or add address information start with *SIOCS*. *SIOC* stands for *socket interface operation command*.

In Chapter 4 we looked at five *protocol-independent* socket operations that modify the addressing information associated with a socket. In this chapter we will see that the command processing is *protocol-dependent*, and that different protocols have different sets of commands associated with these commands.

**Figure 6.11. ioctl function table**



## Ifioctl Function

As shown in [Figure 6.11](#), ifioctl passes protocol the protocol associated with the socket. Control where most of the processing occurs. If the sar

also end up at in\_control. Figure 6.12 repeats t

## Figure 6.12. ifioctl funct

```
if.c
447     default:
448         if (so->so_proto == 0)
449             return (EOPNOTSUPP);
450         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
451                                         cmd, data, ifp));
452     }
453     return (0);
454 }
```

if.c

447-454

The function passes all the relevant data for the request function of the protocol associated with socket, udp\_usrreq is called. Section 23.10 describes how to implement the PRU\_CONTROL code for a new protocol.

```
if (req == PRU_CONTROL)
    return (in_control(so, (int)req, cmd, data, ifp));
```

## in\_control Function

Figure 6.11 shows that control can reach in\_control via a protocol-dependent case in ifioctl. In both cases, in\_control returns. Figure 6.13 shows in\_contrc

## Figure 6.13

```
-----in.c-----  
132 in_control(so, cmd, data, ifp)  
133 struct socket *so;  
134 int cmd;  
135 caddr_t data;  
136 struct ifnet *ifp;  
137 {  
138     struct ifreq *ifr = (struct ifreq *) data;  
139     struct in_ifaddr *ia = 0;  
140     struct ifaddr *ifa;  
141     struct in_ifaddr *oia;  
142     struct in_aliasreq *ifra = (struct in_aliasreq *) data;  
143     struct sockaddr_in oldaddr;  
144     int error, hostIsNew, maskIsNew;  
145     u_long i;  
146     /*  
147      * Find address for this interface, if it exists.  
148      */  
149     if (ifp)  
150         for (ia = in_ifaddr; ia; ia = ia->ia_next)  
151             if (ia->ia_ifp == ifp)  
152                 break;  
153     switch (cmd) {  
154         /* establish preconditions for commands */  
155  
218     }  
219     switch (cmd) {  
220         /* perform the commands */  
221  
326     default:  
327         if (ifp == 0 || ifp->if_ioctl == 0)  
328             return (EOPNOTSUPP);  
329         return ((*ifp->if_ioctl) (ifp, cmd, data));  
330     }  
331     return (0);  
332 }-----in.c-----
```

132-145

so points to the socket on which the ioctl (spec third argument, data, points to the data (secon command. The last argument, ifp, is null (non-i named in the ifreq or in\_aliasreq structures (int to access data as an ifreq or as an in\_aliasreq s

146-152

If ifp points to an ifnet structure, the for loop loops through all the interfaces associated with the interface. If an address is found, the loop exits.

If ifp is null, cmd will not match any of the cases in the first switch. The default case in the second switch processes the command.

153-330

The first switch in in\_control makes sure all the interfaces have an if\_ioctl function. The second switch processes the command. The interface driver's specific processing.

If the default case is executed in the second switch, the interface has an if\_ioctl function, then in\_control calls the interface driver's specific processing.

Net/3 does not define any interface command. A driver for a particular device might define its own command and process it by this case.

331-332

We'll see that many of the cases within the switch statements, in\_control returns 0. Some cases return non-zero values.

We look at the interface ioctl commands in the next section.

- assigning an address, network mask, or destination broadcast address;
- retrieving an address, network mask, destination broadcast address, or subnet mask;
- assigning multiple addresses to an interface;
- deleting an address.

For each group of commands, we describe the preconditions for the statement and then the command processing details.

## Preconditions: SIOCSIFADDR, SIOCSIFNETMASK, SIOCSIFDSTADDR

Figure 6.14 shows the precondition testing for the SIOCSIFDSTADDR command.

**Figure 6.14. in\_control**

```

166     case SIOCSIFADDR:
167     case SIOCSIFNETMASK:
168     case SIOCSIFDSTADDR:
169         if ((so->so_state & SS_PRIV) == 0)
170             return (EPERM);
171
172         if (ifp == 0)
173             panic("in_control");
174         if (ia == (struct in_ifaddr *) 0) {
175             oia = (struct in_ifaddr *)
176                 malloc(sizeof *oia, M_IFADDR, M_WAITOK);
177             if (oia == (struct in_ifaddr *) NULL)
178                 return (ENOBUFS);
179             bzero((caddr_t) oia, sizeof *oia);
180             if (ia = in_ifaddr) {
181                 for (; ia->ia_next; ia = ia->ia_next)
182                     continue;
183                 ia->ia_next = oia;
184             } else
185                 in_ifaddr = oia;
186             ia = oia;
187             if (ifa = ifp->if_addrlist) {
188                 for (; ifa->ifa_next; ifa = ifa->ifa_next)
189                     continue;
190                 ifa->ifa_next = (struct ifaddr *) ia;
191             } else
192                 ifp->if_addrlist = (struct ifaddr *) ia;
193
194             ia->ia_ifa.if_a_addr = (struct sockaddr *) &ia->ia_addr;
195             ia->ia_ifa.if_a_dstaddr
196                 = (struct sockaddr *) &ia->ia_dstaddr;
197             ia->ia_ifa.if_a_netmask
198                 = (struct sockaddr *) &ia->ia_sockmask;
199             ia->ia_sockmask.sin_len = 8;
200             if (ifp->if_flags & IFF_BROADCAST) {
201                 ia->ia_broadaddr.sin_len = sizeof(ia->ia_addr);
202                 ia->ia_broadaddr.sin_family = AF_INET;
203             }
204             ia->ia_ifp = ifp;
205             if (ifp != &loif)
206                 in_interfaces++;
207         }
208     break;

```

in.c

in.c

## Superuser only

166-172

If the socket was not created by a superuser program returns EPERM. If no interface is associated with the socket it can't happen since ifioctl returns -1 if it can't locate an interface.

The SS\_PRIV flag is set by socreate (Figure 1).

Because the test here is against the flag and the root process can create a socket, and give up commands.

## Allocate structure

173-191

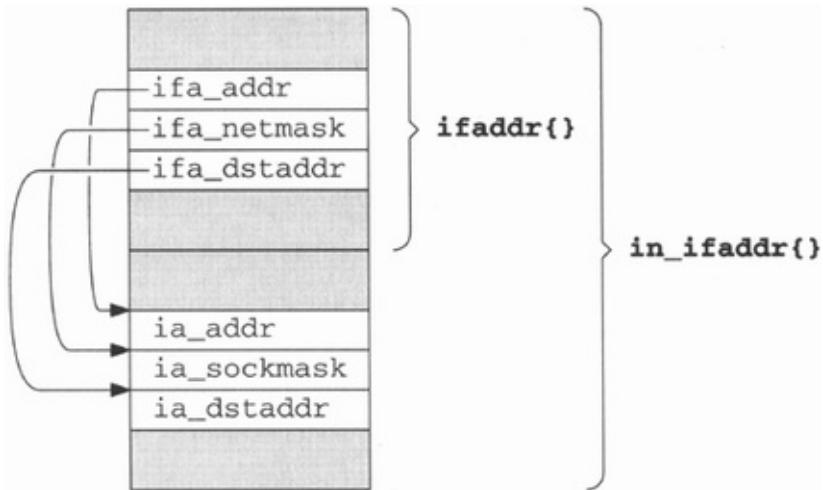
If ia is null, the command is requesting a new interface. It clears it with bzero, and links it into the in\_ifaddr interface.

## Initialize structure

192-201

The next portion of code initializes the in\_ifadd portion of the structure are initialized to point to the in\_ifaddr structure. The function also initializes the ia\_sockmask and ia\_ifindex fields. Figure 6.15 illustrates the in\_ifaddr structure after this initialization.

**Figure 6.15. An in\_ifaddr structure.**



202-206

Finally, in\_control establishes the back pointer

Net/3 counts only nonloopback interfaces in in\_

## Address Assignment: SIOCSIFADDR

The precondition code has ensured that ia points to the SIOCSIFADDR command. [Figure 6.16](#) shows the code for this command.

**Figure 6.16. in\_control**

---

```

259     case SIOCSIFADDR:
260         return (in_ifinit(ifp, ia,
261                         (struct sockaddr_in *) &ifr->ifr_addr, 1));

```

---

259-261

`in_ifinit` does all the work. The IP address inclusion is handled by `in_ifinit`.

## **in\_ifinit Function**

The major steps in `in_ifinit` are:

- copy the address into the structure and info structures,
- discard any routes configured with the previous address,
- establish a subnet mask for the address,
- establish a default route to the attached network,
- join the all-hosts group on the interface.

The code is described in three parts, starting with the first part of the function.

**Figure 6.17. `in_ifinit` function: address assignment**

```
353 in_ifinit(ifp, ia, sin, scrub)                                in.c
354 struct ifnet *ifp;
355 struct in_ifaddr *ia;
356 struct sockaddr_in *sin;
357 int     scrub;
358 {
359     u_long i = ntohl(sin->sin_addr.s_addr);
360     struct sockaddr_in oldaddr;
361     int      s = splimp(), flags = RTF_UP, error, ether_output();
362     oldaddr = ia->ia_addr;
363     ia->ia_addr = *sin;
364     /*
365      * Give the interface a chance to initialize
366      * if this is its first address,
367      * and to validate the address if necessary.
368     */
369     if (ifp->if_ioctl &&
370         (error = (*ifp->if_ioctl) (ifp, SIOCSIFADDR, (caddr_t) ia))) {
371         splx(s);
372         ia->ia_addr = oldaddr;
373         return (error);
374     }
375     if (ifp->if_output == ether_output) { /* XXX: Another Kludge */
376         ia->ia_ifa.ifa_rtrequest = arp_rtrequest;
377         ia->ia_ifa.ifa_flags |= RTP_CLONING;
378     }
379     splx(s);
380     if (scrub) {
381         ia->ia_ifa.ifa_addr = (struct sockaddr *) &oldaddr;
382         in_ifscrub(ifp, ia);
383         ia->ia_ifa.ifa_addr = (struct sockaddr *) &ia->ia_addr;
384     }
```

in.c

## 353-359

The four arguments to `in_ifinit` are: `ifp`, a pointer to the interface structure to be changed; `sin`, a pointer to the route structure; `scrub`, a flag indicating whether existing routes for this interface should be discarded.

## Assign address and notify hardware

## 360-374

`in_ifinit` saves the previous address in `oldaddr` in case the interface has an `if_ioctl` function defined, in `in_ifscrub`. The `if_ioctl` for the sample interfaces are described in the section on sample interfaces.

in\_control returns if an error occurs.

## Ethernet configuration

375-378

For Ethernet devices, arp\_rtrequest is selected flag is set. arp\_rtrequest is described in [Section 19.4](#). As the XXX comment suggests, p drivers.

## Discard previous routes

379-384

If the caller requests that existing routes be sc while in\_ifscrub locates and invalidates any rou the new address is restored.

The section of in\_ifinit shown in [Figure 6.18](#) co

**Figure 6.18. in\_ifinit function**

```

385     if (IN_CLASSA(i))
386         ia->ia_netmask = IN_CLASSA_NET;
387     else if (IN_CLASSB(i))
388         ia->ia_netmask = IN_CLASSB_NET;
389     else
390         ia->ia_netmask = IN_CLASSC_NET;
391     /*
392      * The subnet mask usually includes at least the standard network part,
393      * but may be smaller in the case of supernetting.
394      * If it is set, we believe it.
395     */
396     if (ia->ia_subnetmask == 0) {
397         ia->ia_subnetmask = ia->ia_netmask;
398         ia->ia_sockmask.sin_addr.s_addr = htonl(ia->ia_subnetmask);
399     } else
400         ia->ia_netmask &= ia->ia_subnetmask;
401     ia->ia_net = i & ia->ia_netmask;
402     ia->ia_subnet = i & ia->ia_subnetmask;
403     in_socktrim(&ia->ia_sockmask);

```

- in.c

## Construct network mask and default subnet

385-400

A tentative network mask is constructed in ia\_r  
B, or class C address. If no subnetwork mask is  
ia\_sockmask are initialized to the tentative mas

If a subnet has been specified, in\_ifinit logically  
together to get a new network mask. This oper  
netmask (it can never set the 0 bits, since 0 log  
network mask has fewer 1 bits than would be e

This is called *supernetting* and is described in  
of several class A, class B, or class C networks:  
Volume 1.

An interface is configured by default *without su*  
the same). An explicit request (with SIOCSIFNE

subnetting (or supernetting).

## Construct network and subnetwork numbers

401-403

The network and subnetwork numbers are extracted from the masks. The function `in_socktrim` sets the length `len` by locating the last byte that contains a 1 bit in the mask.

[Figure 6.19](#) shows the last section of `in_ifinit`, which adds the host to a multicast group.

[Figure 6.19. in\\_ifinit function](#)

```

404  /*
405   * Add route for the network.
406   */
407  ia->ia_ifa.ifa_metric = ifp->if_metric;
408  if (ifp->if_flags & IFF_BROADCAST) {
409      ia->ia_broadaddr.sin_addr.s_addr =
410          htonl(ia->ia_subnet | ~ia->ia_subnetmask);
411      ia->ia_netbroadcast.s_addr =
412          htonl(ia->ia_net | ~ia->ia_netmask);
413  } else if (ifp->if_flags & IFF_LOOPBACK) {
414      ia->ia_ifa.ifa_dstaddr = ia->ia_ifa.ifa_addr;
415      flags |= RTF_HOST;
416  } else if (ifp->if_flags & IFF_POINTOPOINT) {
417      if (ia->ia_dstaddr.sin_family != AF_INET)
418          return (0);
419      flags |= RTF_HOST;
420  }
421  if ((error = rtinit(&(ia->ia_ifa), (int) RTM_ADD, flags)) == 0)
422      ia->ia_flags |= IFA_ROUTE;
423  /*
424   * If the interface supports multicast, join the "all hosts"
425   * multicast group on that interface.
426   */
427  if (ifp->if_flags & IFF_MULTICAST) {
428      struct in_addr addr;
429
430      addr.s_addr = htonl(INADDR_ALLHOSTS_GROUP);
431      in_addmulti(&addr, ifp);
432  }
433 }

```

-in.c

## Establish route for host or network

404-422

The next step is to create a route for the network routing metric from the interface to the in\_ifaddr interface. If the interface supports broadcasts, and forces the destination address for loopback interfaces. If a point-to-point link connects the interface to the other end of the link, in\_ifinit returns before

in\_ifinit initializes flags to RTF\_UP and logically connects the interface to the network. rtinit installs a route to the network interface. If rtinit succeeds, the IFA\_ROUTE flag is set in this address.

## Join all-hosts group

423-433

Finally, a multicast capable interface must join in\_addmulti does the work and is described in [Section 6.14](#).

## Network Mask Assignment: SIOCSIFNETMAS

Figure 6.20 shows the processing for the netmask assignment.

### Figure 6.20. in\_control function

```
262     case SIOCSIFNETMASK:  
263         i = ifra->ifra_addr.sin_addr.s_addr;  
264         ia->ia_subnetmask = ntohl(ia->ia_sockmask.sin_addr.s_addr = i);  
265         break;
```

262-265

in\_control extracts the requested netmask from the SIOCSIFNETMASK command and stores it in ia\_subnetmask in hex network byte order.

## Destination Address Assignment: SIOCSIFDSTADDR

For point-to-point interfaces, the address of the SIOCSIFDSTADDR command. Figure 6.14 shows the processing for the destination address assignment.

Figure 6.21.

Figure 6.21. in\_control function

```
-----in.c-----  
236     case SIOCSIFDSTADDR:  
237         if ((ifp->if_flags & IFF_POINTOPOINT) == 0)  
238             return (EINVAL);  
239         oldaddr = ia->ia_dstaddr;  
240         ia->ia_dstaddr = *(struct sockaddr_in *) &ifr->ifr_dstaddr;  
241         if (ifp->if_ioctl && (error = (*ifp->if_ioctl)  
242             (ifp, SIOCSIFDSTADDR, (caddr_t) ia))) {  
243             ia->ia_dstaddr = oldaddr;  
244             return (error);  
245         }  
246         if (ia->ia_flags & IFA_ROUTE) {  
247             ia->ia_ifa.ifa_dstaddr = (struct sockaddr *) &oldaddr;  
248             rtinit(&(ia->ia_ifa), (int) RTM_DELETE, RTF_HOST);  
249             ia->ia_ifa.ifa_dstaddr =  
250                 (struct sockaddr *) &ia->ia_dstaddr;  
251             rtinit(&(ia->ia_ifa), (int) RTM_ADD, RTF_HOST | RTF_UP);  
252         }  
253     break;  
-----in.c-----
```

236-245

Only point-to-point networks have destination address associated with networks. After saving the current destination address, the function informs the interface through the if\_ioctl function.

246-253

If the address has a route previously associated with it, the function removes the old route and a new route to the new destination is installed.

## Retrieving Interface Information

Figure 6.22 shows the precondition processing

commands that return interface information to

**Figure 6.22. in\_com.c**

```
-----in.c-----  
207     case SIOCSIFBRDADDR:  
208         if ((so->so_state & SS_PRIV) == 0)  
209             return (EPERM);  
210         /* FALLTHROUGH */  
  
211     case SIOCGIFADDR:  
212     case SIOCGIFNETMASK:  
213     case SIOCGIFDSTADDR:  
214     case SIOCGIFBRDADDR:  
215         if (ia == (struct in_ifaddr *) 0)  
216             return (EADDRNOTAVAIL);  
217         break;  
-----in.c-----
```

207-217

The broadcast address may only be set through SIOCSIFBRDADDR command and the four SIOC defined for the interface, in which case ia won't null, EADDRNOTAVAIL is returned.

The processing of these five commands (four generic) is shown in Figure 6.23.

**Figure 6.23. in\_cc.c**

```

220     case SIOCGIFADDR:
221         *((struct sockaddr_in *) &ifr->ifr_addr) = ia->ia_addr;
222         break;
223     case SIOCGIFBRDADDR:
224         if ((ifp->if_flags & IFF_BROADCAST) == 0)
225             return (EINVAL);
226         *((struct sockaddr_in *) &ifr->ifr_broadaddr) = ia->ia_broadaddr;
227         break;
228     case SIOCGIFDSTADDR:
229         if ((ifp->if_flags & IFF_POINTOPOINT) == 0)
230             return (EINVAL);
231         *((struct sockaddr_in *) &ifr->ifr_dstaddr) = ia->ia_dstaddr;
232         break;
233     case SIOCGIFNETMASK:
234         *((struct sockaddr_in *) &ifr->ifr_addr) = ia->ia_sockmask;
235         break;

    /* processing for SIOCSIFDSTADDR command (Figure 6.21) */

254     case SIOCSIFBRDADDR:
255         if ((ifp->if_flags & IFF_BROADCAST) == 0)
256             return (EINVAL);
257         ia->ia_broadaddr = *((struct sockaddr_in *) &ifr->ifr_broadaddr;
258         break;

```

-in.c

**220-235**

The unicast address, broadcast address, destination address is available only from a p

**254-258**

The broadcast address is copied from the ifreq

## Multiple IP Addresses per Interface

The SIOCGxxx and SIOCSxxx commands operate on an interface. The first address located by the loop at the end of the list is the primary IP address. Additional addresses are returned in the list.

SIOCAIFADDR command. In fact, SIOCAIFADDI commands do. The ifconfig program uses SIOC an interface.

As noted earlier, having multiple addresses per are renumbered. A fault-tolerant software system assume the IP address of a failed system.

The -alias option to Net/3's ifconfig program passes kernel in an in\_aliasreq structure, shown in Fig

**Figure 6.24.**

```
59 struct in_aliasreq {  
60     char    ifra_name[IFNAMSIZ];    /* interface name, e.g. "en0" */  
61     struct sockaddr_in ifra_addr;  
62     struct sockaddr_in ifra_broadaddr;  
63 #define ifra_dstaddr ifra_broadaddr  
64     struct sockaddr_in ifra_mask;  
65 };
```

59-65

Notice that unlike the ifreq structure, there is no SIOCAIFADDR, the address, broadcast address,

SIOCAIFADDR adds a new address or changes one. SIOCDIFADDR deletes the in\_ifaddr structure from the interface. The precondition processing for the SIOCAIFADDR and SIOCDIFADDR functions in the loop at the start of in\_control (Figure 6.13) has the interface specified in ifra\_name (if it exists).

## Figure 6.25. in\_control func

```
154     case SIOCAIFADDR:  
155     case SIOCDIFADDR:  
156         if (ifra->ifra_addr.sin_family == AF_INET)  
157             for (oia = ia; ia; ia = ia->ia_next) {  
158                 if (ia->ia_ifp == ifp &&  
159                     ia->ia_addr.sin_addr.s_addr ==  
160                     ifra->ifra_addr.sin_addr.s_addr)  
161                     break;  
162             }  
163         if (cmd == SIOCDIFADDR && ia == 0)  
164             return (EADDRNOTAVAIL);  
165     /* FALLTHROUGH to Figure 6.14 */
```

154-165

Because the SIOCDIFADDR code looks only at the ifra pointer, 6.25 works for SIOCAIFADDR (when ifra points to an ifreq structure). The first two fields are identical.

For both commands, the for loop continues the looking for the `in_ifaddr` structure with the same command, `EADDRNOTAVAIL` is returned if the address is not found.

After the loop and the test for the delete command, Figure 6.14. For the add command, the code in was not found that matched the address in the

## **Additional IP Addresses: SIOCAIFADDR**

At this point `ia` points to a new `in_ifaddr` structure that matched the address in the request. The `S`

## Figure 6.26. in\_control function

```
-----in.c-----  
266     case SIOCAIFADDR:  
267         maskIsNew = 0;  
268         hostIsNew = 1;  
269         error = 0;  
270         if (ia->ia_addr.sin_family == AF_INET) {  
271             if (ifra->ifra_addr.sin_len == 0) {  
272                 ifra->ifra_addr = ia->ia_addr;  
273                 hostIsNew = 0;  
274             } else if (ifra->ifra_addr.sin_addr.s_addr ==  
275                         ia->ia_addr.sin_addr.s_addr)  
276                 hostIsNew = 0;  
277         }  
278         if (ifra->ifra_mask.sin_len) {  
279             in_ifscrub(ifp, ia);  
280             ia->ia_sockmask = ifra->ifra_mask;  
281             ia->ia_subnetmask =  
282                 ntohl(ia->ia_sockmask.sin_addr.s_addr);  
283             maskIsNew = 1;  
284         }  
285         if ((ifp->if_flags & IFF_POINTOPOINT) &&  
286             (ifra->ifra_dstaddr.sin_family == AF_INET)) {  
287             in_ifscrub(ifp, ia);  
288             ia->ia_dstaddr = ifra->ifra_dstaddr;  
289             maskIsNew = 1; /* We lie; but the effect's the same */  
290         }  
291         if (ifra->ifra_addr.sin_family == AF_INET &&  
292             (hostIsNew || maskIsNew))  
293             error = in_ifinit(ifp, ia, &ifra->ifra_addr, 0);  
294         if ((ifp->if_flags & IFF_BROADCAST) &&  
295             (ifra->ifra_broadaddr.sin_family == AF_INET))  
296             ia->ia_broadaddr = ifra->ifra_broadaddr;  
297     return (error);  
-----in.c-----
```

266-277

Since SIOCAIFADDR can create a new address or change an existing one, the maskIsNew and hostIsNew flags keep track of whether the address was updated if necessary at the end of the function.

By default, the code assumes that a new IP address is being added (hostIsNew at 1). If the length of the new address is 0, in\_ifinit() changes hostIsNew to 0. If the length is not 0 and the new address request does not contain a new address and hostIsNew is 1, then

278-284

If a netmask is specified in the request, any route in\_control installs the new mask.

285-290

If the interface is a point-to-point interface and in\_scrub discards any routes using the address maskIsNew is set to 1 to force the call to in\_ifir

291-297

If a new address has been configured or a new appropriate changes to support the new configuration in\_ifinit is 0. This indicates that it isn't necessary to care of. Finally, the broadcast address is copied from broadcasts.

## Deleting IP Addresses: SIOCDIFADDR

The SIOCDIFADDR command, which deletes IP addresses. Remember that ia points to the in\_ifaddr struct (from the request).

Figure 6.27. in\_control

```

298     case SIOCDIFADDR:
299         in_ifscrub(ifp, ia);
300         if ((ifa = ifp->if_addrlist) == (struct ifaddr *) ia)
301             /* ia is the first address in the list */
302             ifp->if_addrlist = ifa->ifa_next;
303         else {
304             /* ia is *not* the first address in the list */
305             while (ifa->ifa_next &&
306                   (ifa->ifa_next != (struct ifaddr *) ia))
307                 ifa = ifa->ifa_next;
308             if (ifa->ifa_next)
309                 ifa->ifa_next = ((struct ifaddr *) ia)->ifa_next;
310             else
311                 printf("Couldn't unlink inifaddr from ifp\n");
312         }
313         oia = ia;
314         if (oia == (ia = in_ifaddr))
315             in_ifaddr = ia->ia_next;
316         else {
317             while (ia->ia_next && (ia->ia_next != oia))
318                 ia = ia->ia_next;
319             if (ia->ia_next)
320                 ia->ia_next = oia->ia_next;
321             else
322                 printf("Didn't unlink inifadr from list\n");
323         }
324         IFAFREE(&oia->ia_ifa));
325         break;

```

*in.c*

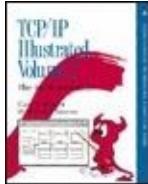
**298-323**

The precondition code arranged for ia to point to routes associated with the address. The first if second if deletes the structure from the Internet.

**324-325**

IFAFREE only releases the structure when the reference count reaches zero.

The additional references would be from entries in the routing table.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

# 6.7 Interface ioctl Processing

We now look at the specific ioctl processing done by each of our sample interfaces in the leioctl, slioctl, and loiectl functions when an address is assigned to the interface.

in\_ifinit is called by the SIOCSIFADDR code in [Figure 6.16](#) and by the SIOCAIFADDR code in [Figure 6.26](#). in\_ifinit always issues the SIOCSIFADDR command through the interface's if\_ioctl function ([Figure 6.17](#)).

## leioctl Function

[Figure 4.31](#) showed SIOCSIFFLAGS

command processing of the LANCE driver. Figure 6.28 shows the SIOCSIFADDR command processing.

## Figure 6.28. ioctl function.

```
if_le.c
614 leioctl(ifp, cmd, data)
615 struct ifnet *ifp;
616 int cmd;
617 caddr_t data;
618 {
619     struct ifaddr *ifa = (struct ifaddr *) data;
620     struct le_softc *le = &le_softc[ifp->if_unit];
621     struct leregl *lerl = le->sc_rl;
622     int s = splimp(), error = 0;
623     switch (cmd) {
624     case SIOCSIFADDR:
625         ifp->if_flags |= IFF_UP;
626         switch (ifa->ifa_addr->sa_family) {
627             case AF_INET:
628                 leinit(ifp->if_unit); /* before arpwhohas */
629                 ((struct arpcom *) ifp)->ac_ipaddr =
630                     IA_SIN(ifa)->sin_addr;
631                 arpwhohas((struct arpcom *) ifp, &IA_SIN(ifa)->sin_addr);
632                 break;
633             default:
634                 leinit(ifp->if_unit);
635                 break;
636         }
637         break;
638
639         /* SIOCSIFFLAGS command (Figure 4.31) */
640         /* SIOCADDMULTI and SIOCDELMULTI commands (Figure 12.31) */
641
642     default:
643         error = EINVAL;
644     }
645     splx(s);
646     return (error);
647 }
```

if\_le.c

614-637

Before processing the command, data is converted to an ifaddr structure pointer and ifp->if\_unit selects the appropriate

le\_softc structure for this request.

The interface is marked as up and the hardware is initialized by leinit. For Internet addresses, the IP address is stored in the arpcom structure and a *gratuitous ARP* for the address is issued. *Gratuitous ARP* is discussed in Section 21.5 and in Section 4.7 of Volume 1.

## Unrecognized commands

672-677

EINVAL is returned for unrecognized commands.

## slioctl Function

The slioctl function ([Figure 6.29](#)) processes the SIOCSIFADDR and SIOCSIFDSTADDR command for the SLIP device driver.

**Figure 6.29. slioctl function:  
SIOCSIFADDR and SIOCSIFDSTADDR  
commands.**

---

```

653 int
654 slioctl(ifp, cmd, data)
655 struct ifnet *ifp;
656 int cmd;
657 caddr_t data;
658 {
659     struct ifaddr *ifa = (struct ifaddr *) data;
660     struct ifreq *ifr;
661     int s = splimp(), error = 0;
662     switch (cmd) {
663     case SIOCSIFADDR:
664         if (ifa->ifa_addr->sa_family == AF_INET)
665             ifp->if_flags |= IFF_UP;
666         else
667             error = EAFNOSUPPORT;
668         break;
669     case SIOCSIFDSTADDR:
670         if (ifa->ifa_addr->sa_family != AF_INET)
671             error = EAFNOSUPPORT;
672         break;
673     /* SIOCADMULTI and SIOCDELMULTI commands (Figure 12.29) */
674     default:
675         error = EINVAL;
676     }
677     splx(s);
678     return (error);
679 }
```

---

—if\_sl.c

## 663-672

For both commands, EAFNOSUPPORT is returned if the address is not an IP address. The SIOCSIFADDR command enables IFF\_UP.

## Unrecognized commands

## 688-693

EINVAL is returned for unrecognized commands.

## ioctl Function

The ioctl function and its implementation of the SIOCSIFADDR command is shown in Figure 6.30.

**Figure 6.30. ioctl function:  
SIOCSIFADDR command.**

```
if_loop.c
135 int
136 ioctl(ifp, cmd, data)
137 struct ifnet *ifp;
138 int cmd;
139 caddr_t data;
140 {
141     struct ifaddr *ifa;
142     struct ifreq *ifr;
143     int error = 0;
144
145     switch (cmd) {
146         case SIOCSIFADDR:
147             ifp->if_flags |= IFF_UP;
148             ifa = (struct ifaddr *) data;
149             /*
150             * Everything else is done at a higher level.
151             */
152             break;
153
154             /* SIOCADDMULTI and SIOCDELMULTI commands (Figure 12.30) */
155
156     default:
157         error = EINVAL;
158     }
159     return (error);
160 }
```

if\_loop.c

135-151

For Internet addresses, ioctl sets IFF\_UP and returns immediately.

## Unrecognized commands

167-171

EINVAL is returned for unrecognized commands.

Notice that for all three example drivers, assigning an address causes the interface to be marked as up (IFF\_UP).

---

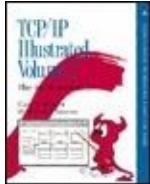
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

## 6.8 Internet Utility Functions

Figure 6.31 lists several functions that manipulate Internet addresses or that rely on the ifnet structures shown in Figure 6.5, usually to discover subnetting information that cannot be obtained from the 32-bit IP address alone. The implementation of these functions consists primarily of traversing data structures and manipulating bit masks. The reader can find these functions in netinet/in.c.

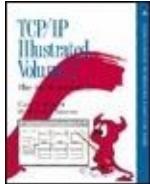
Net/2 had a bug in in\_canforward that permitted loopback addresses to be forwarded. Since most Net/2 systems are configured to recognize only a single loopback address, such as

127.0.0.1, Net/2 systems often forward other addresses in the loopback network (e.g., 127.0.0.2) along the default route.

A telnet to 127.0.0.2 may not do what you expect! ([Exercise 6.6](#))

## Figure 6.31. Internet address functions.

Function	Description
<code>in_netof</code>	Returns network and subnet portions of <i>in</i> . The host bits are set to 0. For class D addresses, returns the class D prefix bits and 0 bits for the multicast group.  <code>u_long in_netof(struct in_addr in);</code>
<code>in_canforward</code>	Returns true if an IP packet addressed to <i>in</i> is eligible for forwarding. Class D and E addresses, loopback network addresses, and addresses with a network number of 0 must not be forwarded.  <code>int in_canforward(struct in_addr in);</code>
<code>in_localaddr</code>	Returns true if the host <i>in</i> is located on a directly connected network. If the global variable <code>subnetsarelocal</code> is nonzero, then subnets of all directly connected networks are also considered local.  <code>int in_localaddr(struct in_addr in);</code>
<code>in_broadcast</code>	Return true if <i>in</i> is a broadcast address associated with the interface pointed to by <i>ifp</i> .  <code>int in_broadcast(struct in_addr in, struct ifnet *ifp);</code>



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

### 6.9 ifnet Utility Functions

Several functions search the data structures shown in [Figure 6.5](#). The functions listed in [Figure 6.32](#) accept addresses for any protocol family, since their argument is a pointer to a sockaddr structure, which contains the address family. Contrast this to the functions in [Figure 6.31](#), each of which takes a 32-bit IP address as an argument. These functions are defined in net/if.c.

**Figure 6.32. ifnet utility functions.**

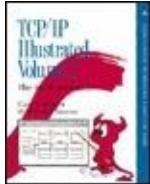
Function	Description
ifa_ifwithaddr	Search the ifnet list for an interface with a unicast or broadcast address of <i>addr</i> . Return a pointer to the matching ifaddr structure or a null pointer if no match is found.  struct ifaddr * <b>ifa_ifwithaddr</b> (struct sockaddr * <i>addr</i> );
ifa_ifwithdstaddr	Search the ifnet list for the interface with a destination address of <i>addr</i> . Return a pointer to the matching ifaddr structure or a null pointer if no match is found.  struct ifaddr * <b>ifa_ifwithdstaddr</b> (struct sockaddr * <i>addr</i> );
ifa_ifwithnet	Search the ifnet list for the address on the same network as <i>addr</i> . Return a pointer to the most specific matching ifaddr structure or a null pointer if no match is found.  struct ifaddr * <b>ifa_ifwithnet</b> (struct sockaddr * <i>addr</i> );
ifa_ifwithaf	Search the ifnet list for the first address in the same address family as <i>addr</i> . Return a pointer to the matching ifaddr structure or a null pointer if no match is found.  struct ifaddr * <b>ifa_ifwithaf</b> (struct sockaddr * <i>addr</i> );
ifaof_ifpforaddr	Search the address list of <i>ifp</i> for the address that matches <i>addr</i> . The order of preference is for an exact match, the destination address on a point-to-point link, an address on the same network, and finally an address in the same address family. Return a pointer to the matching ifaddr structure or a null pointer if no match is found.  struct ifaddr * <b>ifaof_ifpforaddr</b> (struct sockaddr * <i>addr</i> , struct ifnet * <i>ifp</i> );
ifa_ifwithroute	Returns a pointer to the ifaddr structure for the appropriate local interface for the destination ( <i>dst</i> ), and gateway ( <i>gateway</i> ) specified.  struct ifaddr * <b>ifa_ifwithroute</b> (int <i>flags</i> , struct sockaddr * <i>dst</i> , struct sockaddr * <i>gateway</i> )
ifunit	Return a pointer to the ifnet structure associated with <i>name</i> .  struct ifnet * <b>ifunit</b> (char * <i>name</i> );

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 6. IP Addressing

### 6.10 Summary

In this chapter we presented an overview of the IP addressing mechanisms and described interface address structures and protocol address structures that are specialized for IP: the `in_ifaddr` and `sockaddr_in` structures.

We described how interfaces are configured with IP-specific information through the `ifconfig` program and the `ioctl` interface commands.

Finally, we summarized several utility functions that manipulate IP addresses and search the interface data structures.

## Exercises

Why do you think sin\_addr in the  
**6.1** sockaddr\_in structure was originally defined as a structure?

**6.2** ifunit ("s10") returns a pointer to which structure in [Figure 6.5](#)?

Why is the IP address duplicated in  
**6.3** ac\_ipaddr when it is already contained in an ifaddr structure on the interface's address list?

Why do you think IP interface  
**6.4** addresses are accessed through a UDP socket and not a raw IP socket?

Why does in\_socktrim change sin\_len to match the length of the mask  
**6.5** instead of using the standard length of a sockaddr\_in structure?

What happens when the connection request segment from a telnet  
127.0.0.2 command is erroneously  
**6.6** forwarded by a Net/2 system and is  
eventually recognized and accepted by  
a system along the default route?

---

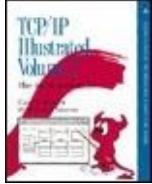
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 7. Domains and Protocols

[Section 7.1. Introduction](#)

[Section 7.2. Code Introduction](#)

[Section 7.3. domain Structure](#)

[Section 7.4. protosw Structure](#)

[Section 7.5. IP domain and protosw Structures](#)

[Section 7.6. pffindproto and pffindtype Functions](#)

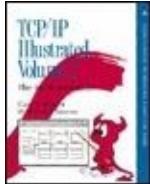
[Section 7.7. pfctlinput Function](#)

[Section 7.8. IP Initialization](#)

[Section 7.9. sysctl System Call](#)

[Section 7.10. Summary](#)

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.1 Introduction

In this chapter we describe the Net/3 data structures that support the concurrent operation of multiple network protocols. We'll use the Internet protocols to illustrate the construction and initialization of these data structures at system initialization time. This chapter presents the necessary background material for our discussion of the IP protocol processing layer, which begins in [Chapter 8](#).

Net/3 groups related protocols into a *domain*, and identifies each domain with a *protocol family* constant. Net/3 also groups protocols by the addressing method they employ. Recall from [Figure 3.19](#) that

address families also have identifying constants. Currently every protocol within a domain uses the same type of address and every address type is used by a single domain. As a result, a domain can be uniquely identified by its protocol family or address family constant. [Figure 7.1](#) lists the protocols and constants that we discuss.

## Figure 7.1. Common protocol and address family constants.

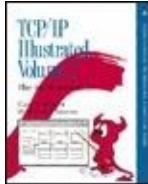
Protocol family	Address family	Protocol
<code>PF_INET</code>	<code>AF_INET</code>	Internet
<code>PF_OSI</code> , <code>PF_ISO</code>	<code>AF_OSI</code> , <code>AF_ISO</code>	OSI
<code>PF_LOCAL</code> , <code>PF_UNIX</code>	<code>AF_LOCAL</code> , <code>AF_UNIX</code>	local IPC (Unix)
<code>PF_ROUTE</code>	<code>AF_ROUTE</code>	routing tables
n/a	<code>AF_LINK</code>	link-level (e.g., Ethernet)

`PF_LOCAL` and `AF_LOCAL` are the primary identifiers for protocols that support communication between processes on the same host and are part of the POSIX.12 standard. Before Net/3, `PF_UNIX` and `AF_UNIX` identified these protocols. The UNIX constants remain for backward compatibility and are used by Net/3 and in this text.

The PF\_UNIX domain supports interprocess communication on a single Unix host. See [[Stevens 1990](#)] for details. The PF\_ROUTE domain supports communication between a process and the routing facilities in the kernel ([Chapter 18](#)). We reference the PF\_OSI protocols occasionally, as some features of Net/3 exist only to support the OSI protocols, but do not discuss them in any detail. Most of our discussions are about the PF\_INET protocols.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

## 7.2 Code Introduction

Two headers and two C files are covered in this chapter. [Figure 7.2](#) describes the four files.

**Figure 7.2. Files discussed in this chapter.**

File	Description
netinet/domain.h netinet/protosw.h	domain structure definition protosw structure definition
netinet/in_proto.c kern/uipc_domain.c	IP domain and protosw structures initialization and search functions

## Global Variables

[Figure 7.3](#) describes several important

global data structures and system parameters that are described in this chapter and referenced throughout Net/3.

### Figure 7.3. Global variables introduced in this chapter.

Variable	Datatype	Description
domains	struct domain *	linked list of domains
inetdomain	struct domain	domain structure for the Internet protocols
inetsw	struct protosw[]	array of protosw structures for the Internet protocols
max_linkhdr	int	see Figure 7.17
max_protohdr	int	see Figure 7.17
max_hdr	int	see Figure 7.17
max_datalen	int	see Figure 7.17

## Statistics

No statistics are collected by the code described in this chapter, but [Figure 7.4](#) shows the statistics table allocated and initialized by the ip\_init function. The only way to look at this table is with a kernel debugger.

### Figure 7.4. Statistics collected in this chapter.

Variable	Datatype	Description
ip_ifmatrix	int [] []	two-dimensional array to count packets routed between any two interfaces

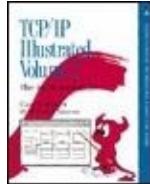
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.3 domain Structure

A protocol domain is represented by a domain structure shown in [Figure 7.5](#).

**Figure 7.5. The domain structure definition.**

```
domain.h
42 struct domain {
43     int     dom_family;          /* AF_XXX */
44     char   *dom_name;
45     void   (*dom_init)();        /* initialize domain data structures */
46             (void);
47     int    (*dom_externalize)(); /* externalize access rights */
48             (struct mbuf *);
49     int    (*dom_dispose)();     /* dispose of internalized rights */
50             (struct mbuf *);
51     struct protosw *dom_protosw, *dom_protoswNPROTOSW;
52     struct domain *dom_next;
53     int    (*dom_rtattach)();    /* initialize routing table */
54             (void **, int);
55     int    dom_rtoffset;        /* an arg to rtattach, in bits */
56     int    dom_maxrtkey;        /* for routing layer */
57 };
```

domain.h

`dom_family` is one of the address family constants (e.g., `AF_INET`) and specifies the addressing employed by the protocols in the domain. `dom_name` is a text name for the domain (e.g., "internet").

The `dom_name` member is not accessed by any part of the Net/3 kernel, but the `fstat(1)` program uses `dom_name` when it formats socket information.

`dom_init` points to the function that initializes the domain. `dom_externalize` and `dom_dispose` point to functions that manage access rights sent across a communication path within the domain. The Unix domain implements this feature to pass file descriptors between processes. The Internet domain does not implement access rights.

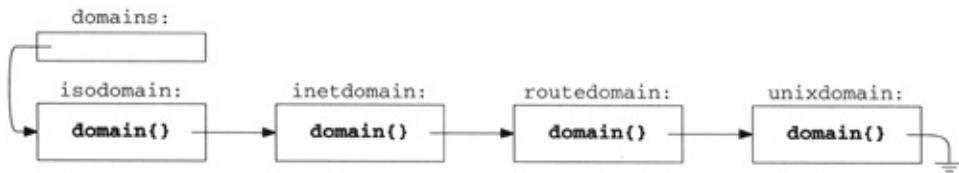
`dom_protosw` and `dom_protoswNPROTOSW` point to the start and end of an array of `protosw` structures. `dom_next` points to the next domain in a linked list of domains supported by the kernel. The linked list of all domains is

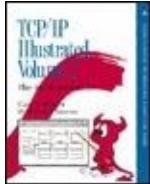
accessed through the global pointer domains.

The next three members, dom\_rtattach, dom\_rtoffset, and dom\_maxrtkey, hold routing information for the domain. They are described in [Chapter 18](#).

[Figure 7.6](#) shows an example domains list.

## Figure 7.6. domains list.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.4 protosw Structure

At compile time, Net/3 allocates and initializes a protosw structure for each protocol in the kernel and groups the structures for all protocols within a single domain into an array. Each domain structure references the appropriate array of protosw structures. A kernel may provide multiple interfaces to the same protocol by providing multiple protosw entries. For example, in [Section 7.5](#) we describe three different entries for the IP protocol.

**Figure 7.7. The protosw structure definition.**

```

57 struct protosw {
58     short pr_type;           /* see (Figure 7.8) */
59     struct domain *pr_domain; /* domain protocol a member of */
60     short pr_protocol;      /* protocol number */
61     short pr_flags;         /* see Figure 7.9 */
62 /* protocol-protocol hooks */
63     void (*pr_input) ();    /* input to protocol (from below) */
64     int  (*pr_output) ();   /* output to protocol (from above) */
65     void (*pr_ctlinput) (); /* control input (from below) */
66     int  (*pr_ctloutput) ();/* control output (from above) */
67 /* user-protocol hook */
68     int  (*pr_usrreq) ();   /* user request from process */
69 /* utility hooks */
70     void (*pr_init) ();     /* initialization hook */
71     void (*pr_fasttimo) (); /* fast timeout (200ms) */
72     void (*pr_slowtimo) (); /* slow timeout (500ms) */
73     void (*pr_drain) ();   /* flush any excess space possible */
74     int  (*pr_sysctl) ();   /* sysctl for protocol */
75 };

```

protosw.h

## 57-61

The first four members in the structure identify and characterize the protocol. `pr_type` specifies the communication semantics of the protocol. [Figure 7.8](#) lists the possible values for `pr_type` and the corresponding Internet protocols.

**Figure 7.8. `pr_type` specifies the protocol's semantics.**

<code>pr_type</code>	Protocol semantics	Internet protocols
<code>SOCK_STREAM</code>	reliable bidirectional byte-stream service	TCP
<code>SOCK_DGRAM</code>	best-effort transport-level datagram service	UDP
<code>SOCK_RAW</code>	best-effort network-level datagram service	ICMP, IGMP, raw IP
<code>SOCK_RDM</code>	reliable datagram service (not implemented)	n/a
<code>SOCK_SEQPACKET</code>	reliable bidirectional record stream service	n/a

`pr_domain` points to the associated domain

structure, pr\_protocol numbers the protocol within the domain, and pr\_flags specifies additional characteristics of the protocol. [Figure 7.9](#) lists the possible values for pr\_flags.

**Figure 7.9. pr\_flags values.**

pr_flags	Description
<i>PR_ATOMIC</i>	each process request maps to a single protocol request
<i>PR_ADDR</i>	protocol passes addresses with each datagram
<i>PR_CONNREQUIRED</i>	protocol is connection oriented
<i>PR_WANTRCVD</i>	notify protocol when a process receives data
<i>PR_RIGHTS</i>	protocol supports access rights

If PR\_ADDR is supported by a protocol, PR\_ATOMIC must also be supported. PR\_ADDR and PR\_CONNREQUIRED are mutually exclusive.

When PR\_WANTRCVD is set, the socket layer notifies the protocol layer when it has passed data from the socket receive buffer to a process (i.e., when more space becomes available in the receive buffer).

PR\_RIGHTS indicates that access right control messages can be passed across

the connection. Access rights require additional support within the kernel to ensure proper cleanup if the receiving process does not consume the messages. Only the Unix domain supports access rights, where they are used to pass descriptors between processes.

[Figure 7.10](#) shows the relationship between the protocol type, the protocol flags, and the protocol semantics.

## Figure 7.10. Protocol characteristics and examples.

pr_type	PR_			Record boundaries?	Reliable?	Example	
	ADDR	ATOMIC	CONNREQUIRED			Internet	Other
SOCK_STREAM			•	none	•	TCP	SPP
SOCK_SEQPACKET		•	•	explicit implicit	•	TP4 SPP	
SOCK_RDM		•	•	implicit	see text		RDP
SOCK_DGRAM	•	•		implicit		UDP	
SOCK_RAW	•	•		implicit		ICMP	

[Figure 7.10](#) does not include the PR\_WANTRCVD or PR\_RIGHTS flags. PR\_WANTRCVD is always set for reliable connection-oriented protocols.

To understand communication semantics

of a protosw entry in Net/3, we must consider the PR\_xxx flags and pr\_type together. In [Figure 7.10](#) we have included two columns ("Record boundaries?" and "Reliable?") to describe the additional semantics that are implicitly specified by pr\_type. [Figure 7.10](#) shows three types of reliable protocols:

- Connection-oriented byte stream protocols such as TCP and SPP (from the XNS protocol family). These protocols are identified by SOCK\_STREAM.
- Connection-oriented stream protocols with record boundaries are specified by SOCK\_SEQPACKET. Within this type of protocol, PR\_ATOMIC indicates whether records are implicitly specified by each output request or are explicitly specified by setting the MSG\_EOR flag on output. TP4 from the OSI protocol family requires explicit record boundaries, and SPP assumes implicit record boundaries.

SPP supports both SOCK\_STREAM

and SOCK\_SEQPACKET semantics.

- The third type of reliable protocol provides a connection-oriented service with implicit record boundaries and is specified by SOCK\_RDM. RDP does not guarantee that records are received in the order that they are sent. RDP is described in [Partridge 1987] and specified by RFC 1151 [Partridge and Hinden 1990].

Two types of unreliable protocols are shown in [Figure 7.10](#):

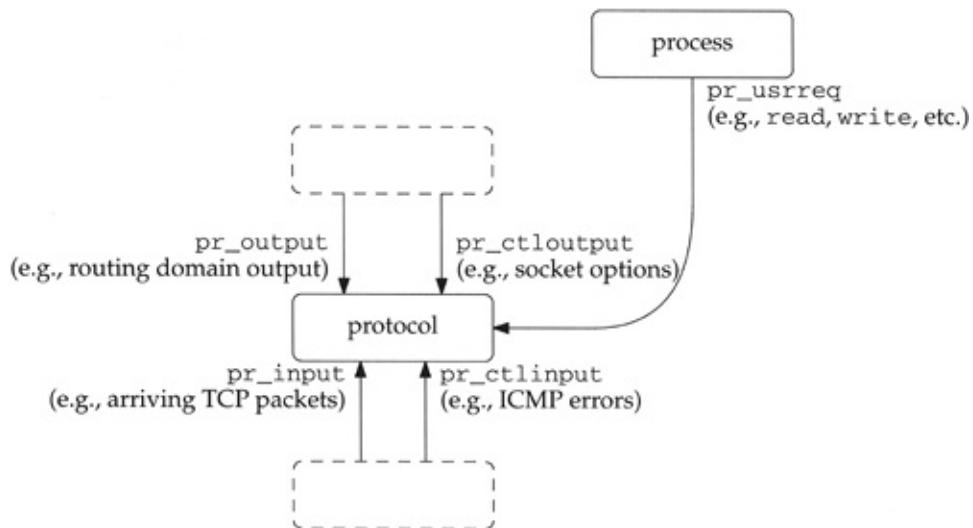
- A transport-level datagram protocol, such as UDP, which includes multiplexing and checksums, is specified by SOCK\_DGRAM.
- A network-level datagram protocol, such as ICMP, which is specified by SOCK\_RAW. In Net/3, only superuser processes may create a SOCK\_RAW socket ([Figure 15.18](#)).

62-68

The next five members are function

pointers providing access to the protocol from other protocols. `pr_input` handles incoming data from a lower-level protocol, `pr_output` handles outgoing data from a higher-level protocol, `pr_ctlinput` handles control information from below, and `pr_ctloutput` handles control information from above. `pr_usrreq` handles all communication requests from a process.

**Figure 7.11. The five main entry points to a protocol.**



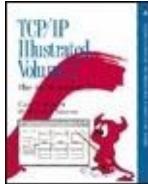
69-75

The remaining five members are utility functions for the protocol. `pr_init` handles

initialization. `pr_fasttimo` and `pr_slowtimo` are called every 200 ms and 500 ms respectively to perform periodic protocol functions, such as updating retransmission timers. `pr_drain` is called by `m_reclaim` when memory is in short supply ([Figure 2.13](#)). It is a request that the protocol release as much memory as possible. `pr_sysctl` provides an interface for the `sysctl(8)` command, a way to modify system-wide parameters, such as enabling packet forwarding or UDP checksum calculations.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.5 IP domain and protosw Structures

The domain and protosw structures for all protocols are declared and initialized statically. For the Internet protocols, the inetsw array contains the protosw structures. [Figure 7.12](#) summarizes the protocol information in the inetsw array. [Figure 7.13](#) shows the definition of the array and the definition of the domain structure for the Internet protocols.

**Figure 7.12. Internet domain protocols.**

inetsw[]	pr_protocol	pr_type	Description	Acronym
0	0	0	Internet Protocol	IP
1	IPPROTO_UDP	SOCK_DGRAM	User Datagram Protocol	UDP
2	IPPROTO_TCP	SOCK_STREAM	Transmission Control Protocol	TCP
3	IPPROTO_RAW	SOCK_RAW	Internet Protocol (raw)	IP (raw)
4	IPPROTO_ICMP	SOCK_RAW	Internet Control Message Protocol	ICMP
5	IPPROTO_IGMP	SOCK_RAW	Internet Group Management Protocol	IGMP
6	0	SOCK_RAW	Internet Protocol (raw, default)	IP (raw)

## Figure 7.13. The Internet domain and protosw structures.

```

39 struct protosw inetsw[] =                                         in_proto.c
40 {
41     {0, &inetdomain, 0, 0,
42      0, ip_output, 0, 0,
43      0,
44      ip_init, 0, ip_slowtimo, ip_drain, ip_sysctl
45    },
46    {SOCK_DGRAM, &inetdomain, IPPROTO_UDP, PR_ATOMIC | PR_ADDR,
47      udp_input, 0, udp_ctlinput, ip_ctloutput,
48      udp_usrreq,
49      udp_init, 0, 0, 0, udp_sysctl
50    },
51    {SOCK_STREAM, &inetdomain, IPPROTO_TCP, PR_CONNREQUIRED | PR_WANTRCV,
52      tcp_input, 0, tcp_ctlinput, tcp_ctloutput,
53      tcp_usrreq,
54      tcp_init, tcp_fasttimo, tcp_slowtimo, tcp_drain,
55    },
56    {SOCK_RAW, &inetdomain, IPPROTO_RAW, PR_ATOMIC | PR_ADDR,
57      rip_input, rip_output, 0, rip_ctloutput,
58      rip_usrreq,
59      0, 0, 0, 0,
60    },
61    {SOCK_RAW, &inetdomain, IPPROTO_ICMP, PR_ATOMIC | PR_ADDR,
62      icmp_input, rip_output, 0, rip_ctloutput,
63      rip_usrreq,
64      0, 0, 0, 0, icmp_sysctl
65    },
66    {SOCK_RAW, &inetdomain, IPPROTO_IGMP, PR_ATOMIC | PR_ADDR,
67      igmp_input, rip_output, 0, rip_ctloutput,
68      rip_usrreq,
69      igmp_init, igmp_fasttimo, 0, 0,
70    },
71    /* raw wildcard */
72    {SOCK_RAW, &inetdomain, 0, PR_ATOMIC | PR_ADDR,
73      rip_input, rip_output, 0, rip_ctloutput,
74      rip_usrreq,
75      rip_init, 0, 0, 0,
76    },
77 };
78 struct domain inetdomain =
79 {AF_INET, "internet", 0, 0, 0,
80  inetsw, &inetsw[sizeof(inetsw) / sizeof(inetsw[0])], 0,
81  rn_inithead, 32, sizeof(struct sockaddr_in));                         in_proto.c

```

Three protosw structures in the inetsw array provide access to IP. The first, inetsw[0], specifies administrative functions for IP and is accessed only by the kernel. The other two entries, inetsw[3] and inetsw[6], are identical except for their pr\_protocol values and provide a *raw* interface to IP. inetsw[3] processes any packets that are received for unrecognized protocols. inetsw[6] is the default raw protocol, which the pffindproto function ([Section 7.6](#)) returns when no other match is found.

In releases before Net/3, packets transmitted through inetsw[3] did not have an IP header prepended. It was the responsibility of the process to construct the correct header. Packets transmitted through inetsw[6] had an IP header prepended by the kernel. 4.3BSD Reno introduced the IP\_HDRINCL socket option ([Section 32.8](#)), so the distinction between inetsw[3] and inetsw[6] is no longer relevant.

The raw interface allows a process to send

and receive IP packets without an intervening transport protocol. One use of the raw interface is to implement a transport protocol outside the kernel. Once the protocol has stabilized, it can be moved into the kernel to improve its performance and availability to other processes. Another use is for diagnostic tools such as traceroute, which uses the raw IP interface to access IP directly. [Chapter 32](#) discusses the raw IP interface. [Figure 7.14](#) summarizes the IP protosw structures.

**Figure 7.14. The IP inetsw entries.**

protosw	inetsw[0]	inetsw[3 and 6]	Description
pr_type	0	SOCK_RAW	IP provides raw packet services
pr_domain	&inetdomain	&inetdomain	both protocols are part of the Internet domain
pr_protocol	0	IPPROTO_RAW or 0	both IPPROTO_RAW (255) and 0 are reserved (RFC 1700) and should never appear in an IP datagram
pr_flags	0	PR_ATOMIC/PR_ADDR	socket layer flags, not used by IP
pr_input	null	rip_input	receive unrecognized datagrams from IP, ICMP, or IGMP
pr_output	ip_output	rip_output	prepare and send datagrams to the IP and hardware layers respectively
pr_ctlinput	null	null	not used by IP
pr_ctloutput	null	rip_ctloutput	respond to configuration requests from a process
pr_usrreq	null	rip_usrreq	respond to protocol requests from a process
pr_init	ip_init	null or rip_init	ip_init does all initialization
pr_fasttimo	null	null	not used by IP
pr_slowtimo	ip_slowtimo	null	slow timeout is used by IP reassembly algorithm
pr_drain	ip_drain	null	release memory if possible
pr_sysctl	ip_sysctl	null	modify systemwide parameters

The domain structure for the Internet protocols is shown at the end of [Figure 7.13](#). The Internet domain uses AF\_INET style addressing, has a text name of "internet", has no initialization or control-message functions, and has its protosw structures in the inetsw array.

The routing initialization function for the Internet protocols is rn\_inithead. The offset of an IP address from the beginning of a sockaddr\_in structure is 32 bits and the size of the structure is 16 bytes ([Figure 18.27](#)).

The only difference between inetsw[3] and inetsw[6] is in their pr\_protocol numbers and the initialization function rip\_init, which is defined only in inetsw[6] so that it is called only once during initialization.

## domaininit Function

At system initialization time ([Figure 3.23](#)), the kernel calls domaininit to link the domain and protosw structures. domaininit is shown in

Figure 7.15.

## Figure 7.15. domaininit function.

```
37 /* simplifies code in domaininit */
38 #define ADDDOMAIN(x)    ( \
39     extern struct domain __CONCAT(x, domain); \
40     __CONCAT(x, domain).dom_next = domains; \
41     domains = &__CONCAT(x, domain); \
42 )
43 domaininit()
44 {
45     struct domain *dp;
46     struct protosw *pr;
47     /* The C compiler usually defines unix. We don't want to get
48      * confused with the unix argument to ADDDOMAIN
49      */
50 #undef unix
51     ADDDOMAIN(unix);
52     ADDDOMAIN(route);
53     ADDDOMAIN/inet);
54     ADDDOMAIN(iso);
55     for (dp = domains; dp; dp = dp->dom_next) {
56         if (dp->dom_init)
57             (*dp->dom_init) ();
58         for (pr = dp->dom_protosw; pr < dp->dom_protosw+NPROTOSW; pr++)
59             if (pr->pr_init)
60                 (*pr->pr_init) ();
61     }
62     if (max_linkhdr < 16)          /* XXX */
63         max_linkhdr = 16;
64     max_hdr = max_linkhdr + max_protohdr;
65     max_dataLEN = MHLEN - max_hdr;
66     timeout(pf_fasttimo, (void *) 0, 1);
67     timeout(pf_slowtimo, (void *) 0, 1);
68 }  
————— uipc_domain.c
```

37-42

The ADDDOMAIN macro declares and links a single domain structure. For example, ADDDOMAIN (unix) expands to

extern struct domain unixdomain;

```
unixdomain.dom_next = domains;  
domains = &unixdomain;
```

The `__CONCAT` macro is defined in `sys/defs.h` and concatenates two symbols. For example, `__CONCAT (unix, domain)` produces `unixdomain`.

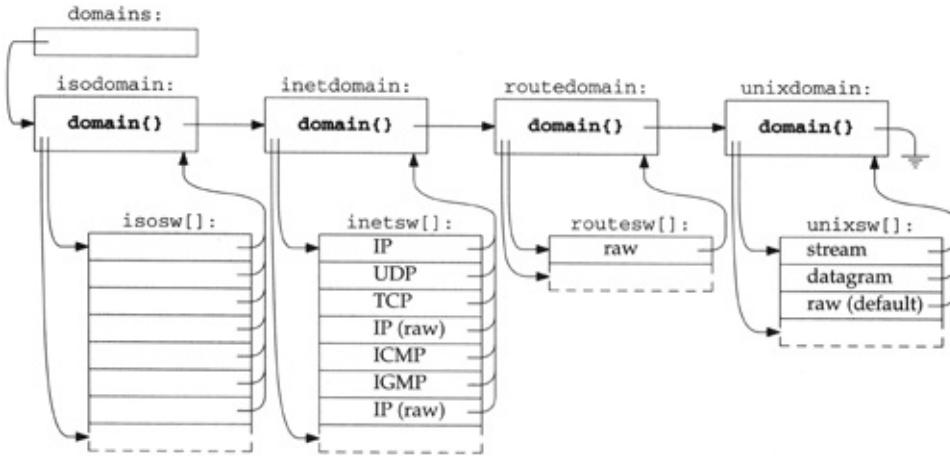
43-54

`domaininit` constructs the list of domains by calling `ADDDOMAIN` for each supported domain.

Since the symbol `unix` is often predefined by the C preprocessor, Net/3 explicitly undefines it here so `ADDDOMAIN` works correctly.

[Figure 7.16](#) shows the linked domain and `protosw` structures in a kernel configured to support the Internet, Unix, and OSI protocol families.

**Figure 7.16. The domain list and `protosw` arrays after initialization.**



55-61

The two nested for loops locate every domain and protocol in the kernel and call the initialization functions `dom_init` and `pr_init` if they are defined. For the Internet protocols, the following functions are called ([Figure 7.13](#)): `ip_init`, `udp_init`, `tcp_init`, `igmp_init`, and `rip_init`.

62-65

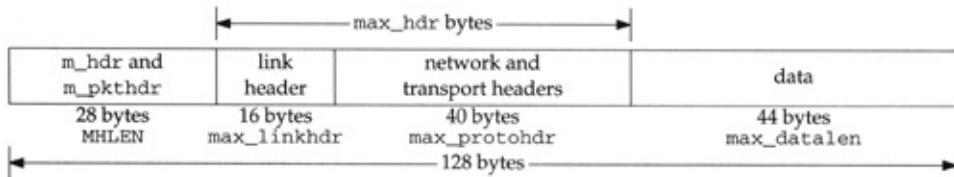
The parameters computed in `domaininit` control the layout of packets in the mbufs to avoid extraneous copying of data. `max_linkhdr` and `max_protohdr` are set during protocol initialization. `domaininit` enforces a lower bound of 16 for `max_linkhdr`. The value of 16 leaves room for a 14-byte Ethernet header ending on a 4-byte

boundary. Figures 7.17 and 7.18 list the parameters and typical values.

**Figure 7.17. Parameters used to minimize copying of protocol data.**

Variable	Value	Description
max_linkhdr	16	maximum number of bytes added by link layer
max_protohdr	40	maximum number of bytes added by network and transport layers
max_hdr	56	$\text{max\_linkhdr} + \text{max\_protohdr}$
max_datalen	44	number of data bytes available in packet header mbuf after accounting for the link and protocol headers

**Figure 7.18. Mbuf and associated maximum header lengths.**



`max_protohdr` is a soft limit that measures the expected protocol header size. In the Internet domain, the IP and TCP headers are usually 20 bytes in length but both can be up to 60 bytes. The penalty for exceeding `max_protohdr` is the time required to push back the data to make room for the larger than expected protocol

header.

66-68

domaininit initiates pfslowtimo and pffasttimo by calling timeout. The third argument specifies when the kernel should call the functions, in this case in 1 clock tick. Both functions are shown in [Figure 7.19](#).

## Figure 7.19. pfslowtimo and pffasttimo functions.

```
153 void                                     uipc_domain.c
154 pfslowtimo(arg)
155 void *arg;
156 {
157     struct domain *dp;
158     struct protosw *pr;
159     for (dp = domains; dp; dp = dp->dom_next)
160         for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
161             if (pr->pr_slowtimo)
162                 (*pr->pr_slowtimo) ();
163     timeout(pfslowtimo, (void *) 0, hz / 2);
164 }

165 void
166 pffasttimo(arg)
167 void *arg;
168 {
169     struct domain *dp;
170     struct protosw *pr;
171     for (dp = domains; dp; dp = dp->dom_next)
172         for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
173             if (pr->pr_fasttimo)
174                 (*pr->pr_fasttimo) ();
175     timeout(pffasttimo, (void *) 0, hz / 5);
176 }
```

153-176

These nearly identical functions use two for loops to call the pr\_slowtimo or pr\_fasttimo function for each protocol, if they are defined. The functions schedule themselves to be called 500 and 200 ms later by calling timeout, which we described with [Figure 3.43](#).

---

## Chapter 7. Domains and Protocols

### 7.6 pffindproto and pffindtype Functions

The pffindproto and pffindtype functions look up a domain by protocol (e.g., IPPROTO\_TCP) or by type (e.g., SOCK\_STREAM). As described in [Chapter 15](#), these functions are called to locate the appropriate socket entry when a process creates a socket.

69-84

pffindtype performs a linear search of domains and then searches the protocols within the domain for a match to the specified type.

85-107

pffindproto searches domains exactly as pffindtype does. It takes the family, type, and protocol specified by the calling function and finds a (protocol, type) match within the specified domain. If the type is SOCK\_RAW, and the domain has a default protocol, then the protocol is set to the default value.

(pr\_protocol equals 0), then pffindproto selects instead of failing completely. For example, a ca

**Figure 7.20. pffindproto and pffind**

```
69 struct protosw *
70 pffindtype(family, type)
71 int      family, type;
72 {
73     struct domain *dp;
74     struct protosw *pr;

75     for (dp = domains; dp; dp = dp->dom_next)
76         if (dp->dom_family == family)
77             goto found;
78     return (0);
79 found:
80     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
81         if (pr->pr_type && pr->pr_type == type)
82             return (pr);
83     return (0);
84 }

85 struct protosw *
86 pffindproto(family, protocol, type)
87 int      family, protocol, type;
88 {
89     struct domain *dp;
90     struct protosw *pr;
91     struct protosw *maybe = 0;

92     if (family == 0)
93         return (0);
94     for (dp = domains; dp; dp = dp->dom_next)
95         if (dp->dom_family == family)
96             goto found;
97     return (0);
98 found:
99     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++) {
100         if ((pr->pr_protocol == protocol) && (pr->pr_type == type))
101             return (pr);

102         if (type == SOCK_RAW && pr->pr_type == SOCK_RAW &&
103             pr->pr_protocol == 0 && maybe == (struct protosw *) 0)
104             maybe = pr;
105     }
106     return (maybe);
107 }
```

uipc\_domain.c

pffindproto(PF\_INET, 27, SOCK\_RF

returns a pointer to inetsw[6], the default raw socket. This does not include support for protocol 27. With a

could implement protocol 27 services on its own and manage the sending and receiving of the IP packets.

Protocol 27 is reserved for the Reliable Datagram Protocol.

Both functions return a pointer to the protosw structure for the specified protocol, or a null pointer if they don't find a matching protocol.

## Example

We'll see in [Section 15.6](#) that when an application calls

```
socket(PF_INET, SOCK_STREAM, 0);
```

`pffindtype` gets called as

```
pffindtype(PF_INET, SOCK_STREAM);
```

[Figure 7.12](#) shows that `pffindtype` will return a pointer to TCP because TCP is the first SOCK\_STREAM protocol in the `proto_table`.

```
socket(PF_INET, SOCK_DGRAM, 0);
```

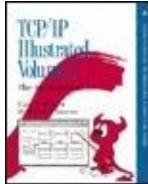
leads to

```
pffindtype(PF_INET, SOCK_DGRAM);
```

which returns a pointer to UDP in inetsw[1].

---

Team-Fly



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.7 pfctlinput Function

The pfctlinput function issues a control request to every protocol in every domain. It is used when an event that may affect every protocol occurs, such as an interface shutdown or routing table change. ICMP calls pfctlinput when an ICMP redirect message arrives (Figure 11.14), since the redirect can affect all the Internet protocols (e.g., UDP and TCP).

**Figure 7.21. pfctlinput function.**

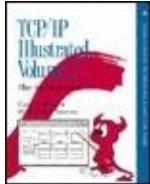
```
142 pfctlinput(cmd, sa)                                uipc_domain.c
143 int          cmd;
144 struct sockaddr *sa;
145 {
146     struct domain *dp;
147     struct protosw *pr;
148     for (dp = domains; dp; dp = dp->dom_next)
149         for (pr = dp->dom_protosw; pr < dp->dom_protosw+NPROTOSW; pr++)
150             if (pr->pr_ctlinput)
151                 (*pr->pr_ctlinput) (cmd, sa, (caddr_t) 0);
152 }
```

---

```
uipc_domain.c
```

## 142-152

The two nested for loops locate every protocol in every domain. pfctlinput issues the protocol control command specified by cmd by calling each protocol's pr\_ctlinput function. For UDP, udp\_ctlinput is called and for TCP, tcp\_ctlinput is called.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.8 IP Initialization

As shown in [Figure 7.13](#), the Internet domain does not have an initialization function but the individual Internet protocols do. For now, we look only at `ip_init`, the IP initialization function. In [Chapters 23 and 24](#) we discuss the UDP and TCP initialization functions. Before we can discuss the code, we need to describe the `ip_protox` array.

### Internet Transport Demultiplexing

A network-level protocol like IP must demultiplex incoming datagrams and deliver them to the appropriate transport-

level protocols. To do this, the appropriate protosw structure must be derived from a protocol number present in the datagram. For the Internet protocols, this is done by the ip\_protox array.

The index into the ip\_protox array is the protocol value from the IP header (ip\_p, [Figure 8.8](#)). The entry selected is the index of the protocol in the inetsw array that processes the datagram. For example, a datagram with a protocol number of 6 is processed by inetsw[2], the TCP protocol. The kernel constructs ip\_protox during protocol initialization, described in [Figure 7.23](#).

## ip\_init Function

The ip\_init function is called by domaininit ([Figure 7.15](#)) at system initialization time.

71-78

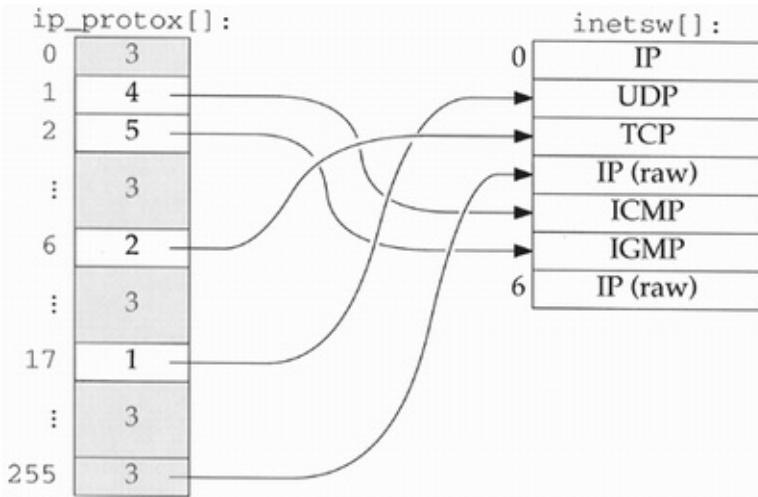
pffindproto returns a pointer to the raw protocol (inetsw[3], [Figure 7.14](#)). Net/3 panics if the raw protocol cannot be

located, since it is a required part of the kernel. If it is missing, the kernel has been misconfigured. IP delivers packets that arrive for an unknown transport protocol to this protocol where they may be handled by a process outside the kernel.

79-85

The next two loops initialize the ip\_protox array. The first loop sets each entry in the array to pr, the index of the default protocol (3 from [Figure 7.22](#)). The second loop examines each protocol in inetsw (other than the entries with protocol numbers of 0 or IPPROTO\_RAW) and sets the matching entry in ip\_protox to refer to the appropriate inetsw entry. Therefore, pr\_protocol in each protosw structure must be the protocol number expected to appear in the incoming datagram.

**Figure 7.22. The ip\_protox array maps the protocol number to an entry in the inetsw array.**



86-92

`ip_init` initializes the IP reassembly queue, `ipq` ([Section 10.6](#)), seeds `ip_id` from the system clock, and sets the maximum size of the IP input queue (`ipintrq`) to 50 (`ipqmaxlen`). `ip_id` is set from the system clock to provide a random starting point for datagram identifiers ([Section 10.6](#)). Finally, `ip_init` allocates a two-dimensional array, `ip_ifmatrix`, to count packets routed between the interfaces in the system.

There are many variables within Net/3 that may be modified by a system administrator. To allow these variables to be changed at run time and without recompiling the kernel, the default value represented by a constant

(IFQ\_MAXLEN in this case) is assigned to a variable (ipq maxlen) at compile time. A system administrator can use a kernel debugger such as adb to change ipq maxlen and reboot the kernel with the new value. If Figure 7.23 used IFQ\_MAXLEN directly, it would require a recompile of the kernel to change the limit.

## Figure 7.23. ip\_init function.

```
71 void  
72 ip_init()  
73 {  
74     struct protosw *pr;  
75     int i;  
76     pr = pffindproto(PF_INET, IPPROTO_RAW, SOCK_RAW);  
77     if (pr == 0)  
78         panic("ip_init");  
79     for (i = 0; i < IPPROTO_MAX; i++)  
80         ip_protox[i] = pr - inetsw;  
81     for (pr = inetdomain.dom_protosw;  
82          pr < inetdomain.dom_protoswNPROTOSW; pr++)  
83         if (pr->pr_domain->dom_family == PF_INET &&  
84             pr->pr_protocol && pr->pr_protocol != IPPROTO_RAW)  
85             ip_protox[pr->pr_protocol] = pr - inetsw;  
86     ipq.next = ipq.prev = &ipq;  
87     ip_id = time.tv_sec & 0xffff;  
88     ipintrq.ifq_maxlen = ipq maxlen;  
89     i = (if_index + 1) * (if_index + 1) * sizeof(u_long);  
90     ip_ifmatrix = (u_long *) malloc(i, M_RTABLE, M_WAITOK);  
91     bzero((char *) ip_ifmatrix, i);  
92 }
```

ip\_input.c

## Chapter 7. Domains and Protocols

---

### 7.9 sysctl System Call

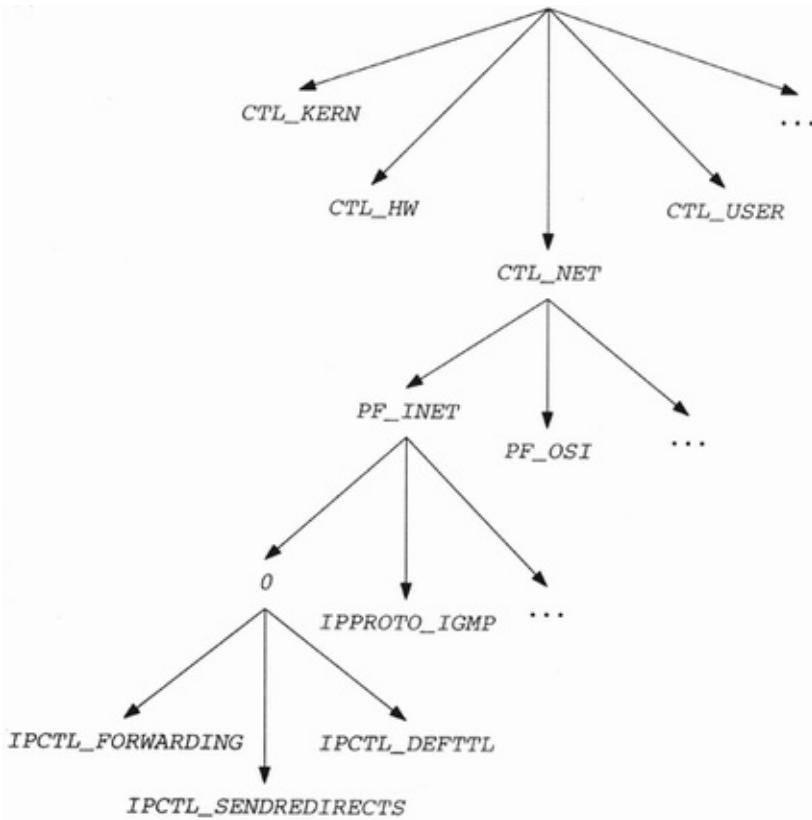
The sysctl system call accesses and modifies NIS+ parameters. A program can modify the parameters through the sysctl(*name*, *namelen*, *oldp*, *newp*) call. The *name* argument is a list of integers and has an associated type. The *namelen* argument is the length of the *name* array.

```
int sysctl (int *name, u_int namelen,
            size_t newlen);
```

The *name* argument points to an array containing *namelen* integers. The old value is passed back in *oldp*, and the new value is passed in the area pointed to by *newp*.

Figure 7.24 summarizes the organization of the sysctl system call.

**Figure 7**



In Figure 7.24, the full name for the IP forward

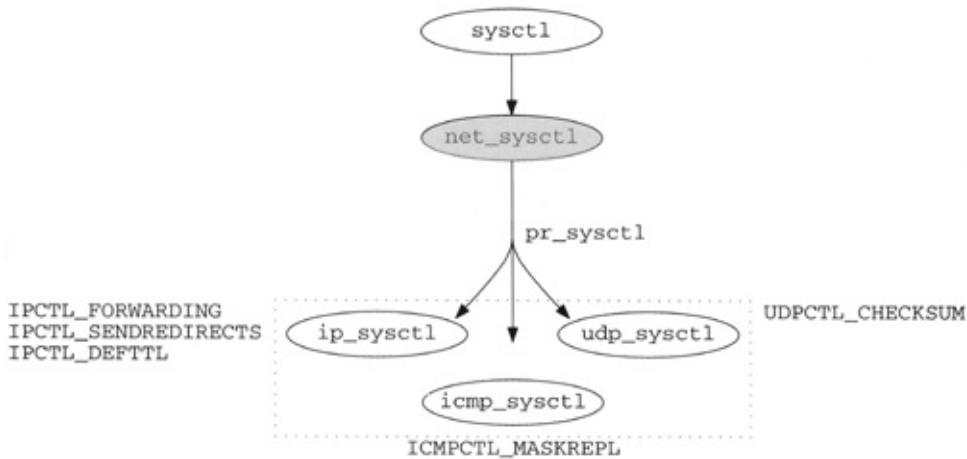
*CTL\_NET, PF\_INET, 0, IPCTL\_FORWARDING*

with the four integers stored in an array.

## net\_sysctl Function

Each level of the sysctl naming scheme is handled by functions that handle the Internet parameters.

**Figure 7.25. sysctl function flowchart**



The top-level names are processed by `sysctl`. The `net_sysctl` function which dispatches control based on the family and protocol's protosw entry.

`sysctl` is implemented in the kernel by the `_sysctl` function. This function contains code to move the `sysctl` arguments to the appropriate function to process the arguments.

**Figure 7.26** shows the `net_sysctl` function.

**Figure 7.26**

```

108 net_sysctl(name, namelen, oldp, oldlenp, newp, newlen, p) ————— uipc_domain.c
109 int     *name;
110 u_int   namelen;
111 void    *oldp;
112 size_t  *oldlenp;
113 void    *newp;
114 size_t  newlen;
115 struct proc *p;
116 {
117     struct domain *dp;
118     struct protosw *pr;
119     int      family, protocol;
120
121     /*
122      * All sysctl names at this level are nonterminal;
123      * next two components are protocol family and protocol number,
124      * then at least one additional component.
125      */
126     if (namelen < 3)
127         return (EISDIR);           /* overloaded */
128     family = name[0];
129     protocol = name[1];
130
131     if (family == 0)
132         return (0);
133     for (dp = domains; dp; dp = dp->dom_next)
134         if (dp->dom_family == family)
135             goto found;
136     return (ENOPROTOOPT);
137
138 found:
139     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
140         if (pr->pr_protocol == protocol && pr->pr_sysctl)
141             return ((*pr->pr_sysctl) (name + 2, namelen - 2,
142                                         oldp, oldlenp, newp, newlen));
143
144     return (ENOPROTOOPT);
145 }

```

————— uipc\_domain.c

## 108-119

The arguments to `net_sysctl` are the same as `tl` which points to the current process structure.

## 120-134

The next two integers in the name are taken to specified in the domain and protosw structures. specified, the for loop searches the domain list match is not found.

## 135-141

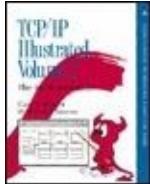
Within a matching domain, the second for loop function defined. When a match is found, the reference to the protocol is returned. Notice that name is advanced to pass the matching protocol is found, ENOPROTOOPT is returned.

Figure 7.27 shows the pr\_sysctl functions defined in the kernel.

**Figure 7.27. pr\_sysctl functions**

pr_protocol	inetsw[]	pr_sysctl	Description	Reference
0	0	<i>ip_sysctl</i>	IP	Section 8.9
<i>IPPROTO_UDP</i>	1	<i>udp_sysctl</i>	UDP	Section 23.11
<i>IPPROTO_ICMP</i>	4	<i>icmp_sysctl</i>	ICMP	Section 11.14

In the routing domain, pr\_sysctl points to the *ip\_sysctl* function.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 7. Domains and Protocols

### 7.10 Summary

We started this chapter by describing the domain and protosw structures that describe and group protocols within the Net/3 kernel. We saw that all the protosw structures for a domain are allocated in an array at compile time and that inetdomain and the inetsw array describe the Internet protocols. We took a closer look at the three inetsw entries that describe the IP protocol: one for the kernel's use and the other two for access to IP by a process.

At system initialization time domaininit links the domains into the domains list, calls the domain and protocol initialization functions, and calls the fast and slow

timeout functions.

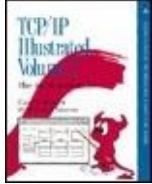
The two functions pffindproto and pffindtype search the domain and protocol lists by protocol number or type, pfctlinput sends a control command to every protocol.

Finally we described the IP initialization procedure including transport demultiplexing by the ip\_protox array.

## Exercises

**7.1** What call to the pffindproto returns a pointer to inetsw[6] ?





[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Chapter 8. IP: Internet Protocol

Section 8.1. Introduction

Section 8.2. Code Introduction

Section 8.3. IP Packets

Section 8.4. Input Processing: ipintr Function

Section 8.5. Forwarding: ip\_forward Function

Section 8.6. Output Processing:  
ip\_output Function

Section 8.7. Internet Checksum:  
in\_cksum Function

Section 8.8. setsockopt and getsockopt  
System Calls

Section 8.9. ip\_sysctl Function

Section 8.10. Summary

---

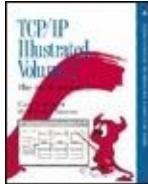
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

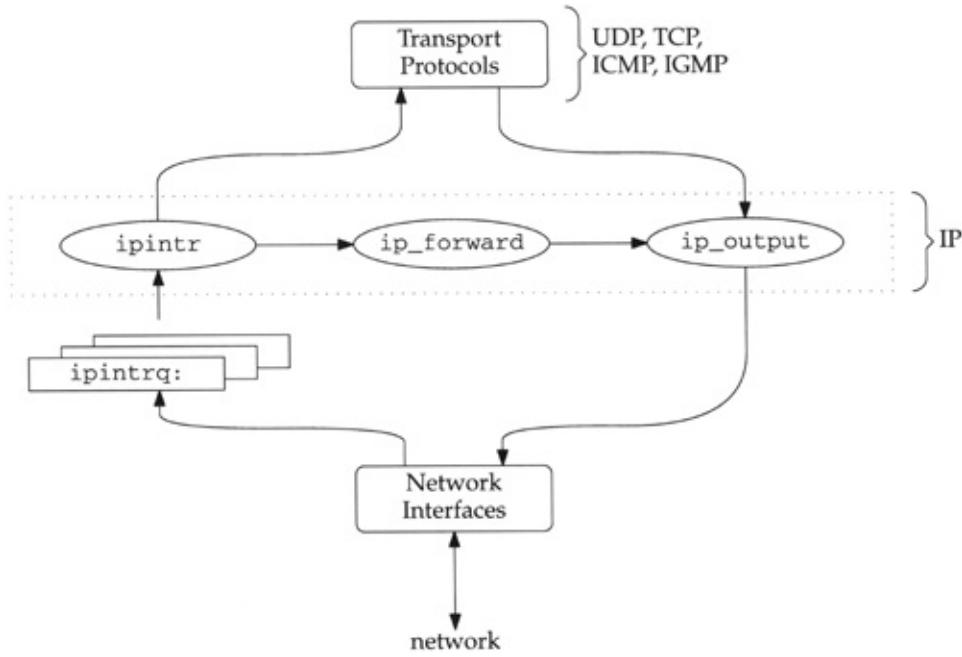
## Chapter 8. IP: Internet Protocol

### 8.1 Introduction

In this chapter we describe the structure of an IP packet and the basic IP processing including input, forwarding, and output. We assume that the reader is familiar with the basic operation of the IP protocol. For more background on IP, see Chapters 3, 9 and 12 of Volume 1. RFC 791 [[Postel 1981a](#)] is the official specification for IP. RFC 1122 [[Braden 1989a](#)] contains clarifications of RFC 791.

In [Chapter 9](#) we discuss option processing and in [Chapter 10](#) we discuss fragmentation and reassembly. [Figure 8.1](#) illustrates the general organization of the IP layer.

## Figure 8.1. IP layer processing.



We saw in [Chapter 4](#) how network interfaces place incoming IP packets on the IP input queue, `ipintrq` and how they schedule a software interrupt. Since hardware interrupts have a higher priority than software interrupts, several packets may be placed on the queue before a software interrupt occurs. During software interrupt processing, the `ipintr` function removes and processes packets from `ipintrq` until the queue is empty. At the final destination, IP reassembles packets into datagrams and passes the datagrams directly to the appropriate transport-level

protocol by a function call. If the packets haven't reached their final destination, IP passes them to `ip_forward` if the host is configured to act as a router. The transport protocols and `ip_forward` pass outgoing packets to `ip_output`, which completes the IP header, selects an output interface, and fragments the outgoing packet if necessary. The resulting packets are passed to the appropriate network interface output function.

When an error occurs, IP discards the packet and under certain conditions may send an error message to the source of the original packet. These messages are part of ICMP ([Chapter 11](#)). Net/3 sends ICMP error messages by calling `icmp_error`, which accepts an mbuf containing the erroneous packet, the type of error found, and an option code that provides additional information depending on the type of error. In this chapter, we describe why and when IP sends ICMP messages, but we postpone a detailed discussion of ICMP itself until [Chapter 11](#).

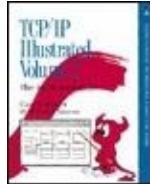
---

**Team-Fly** 

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.2 Code Introduction

Two headers and three C files are discussed in this chapter.

#### Figure 8.2. Files discussed in this chapter.

File	Description
net/route.h	route entries
netinet/ip.h	IP header structure
netinet/ip_input.c	IP input processing
netinet/ip_output.c	IP output processing
netinet/in_cksum.c	Internet checksum algorithm

### Global Variables

Several global variables appear in the IP

processing code. They are described in [Figure 8.3](#).

## Figure 8.3. Global variables introduced in this chapter.

Variable	Datatype	Description
in_ifaddr	struct in_ifaddr *	IP address list
ip_defttl	int	default TTL for IP packets
ip_id	int	last ID assigned to an outgoing IP packet
ip_protos	int[]	demultiplexing array for IP packets
ipforwarding	int	should the system forward IP packets?
ipforward_rt	struct route	cache of most recent forwarded route
ipintrq	struct ifqueue	IP input queue
ipqmaxlen	int	maximum length of IP input queue
ipsendredirects	int	should the system send ICMP redirects?
ipstat	struct ipstat	IP statistics

## Statistics

All the statistics collected by IP are found in the ipstat structure described by [Figure 8.4](#). [Figure 8.5](#) shows some sample output of these statistics, from the netstat -s command. These statistics were collected after the host had been up for 30 days.

## Figure 8.4. Statistics collected in this chapter.

ipstat member	Description	Used by SNMP
ips_badhlen	#packets with invalid IP header length	•
ips_badlen	#packets with inconsistent IP header and IP data lengths	•
ips_badoptions	#packets discovered with errors in option processing	•
ips_badsum	#packets with bad checksum	•
ips_badvers	#packets with an IP version other than 4	•
ips_cantforward	#packets received for unreachable destination	•
ips_delivered	#datagrams delivered to upper level	•
ips_forward	#packets forwarded	•
ips_fragdropped	#fragments dropped (duplicates or out of space)	•
ips_fragments	#fragments received	•
ips_fragtimeout	#fragments timed out	•
ips_noproto	#packets with an unknown or unsupported protocol	•
ips_reassembled	#datagrams reassembled	•
ips_tooshort	#packets with invalid data length	•
ips_toosmall	#packets too small to contain IP packet	•
ips_total	total #packets received	•
ips_cantfrag	#packets discarded because of the don't fragment bit	•
ips_fragmented	#datagrams successfully fragmented	•
ips_localout	#datagrams generated at system (i.e., not forwarded)	•
ips_noroute	#packets discarded—no route to destination	•
ips_odropped	#packets dropped because of resource shortages	•
ips_ofragments	#fragments created for output	•
ips_rawout	total #raw ip packets generated	
ips_redirectsent	#redirect messages sent	

**Figure 8.5. Sample IP statistics.**

netstat -s output	ipstat members
27,881,978 total packets received	ips_total
6 bad header checksums	ips_badsum
9 with size smaller than minimum	ips_tooshort
14 with data size < data length	ips_toosmall
0 with header length < data size	ips_badhlen
0 with data length < header length	ips_badlen
0 with bad options	ips_badoptions
0 with incorrect version number	ips_badvers
72,786 fragments received	ips_fragments
0 fragments dropped (dup or out of space)	ips_fragdropped
349 fragments dropped after timeout	ips_fragtimeout
16,557 packets reassembled ok	ips_reassembled
27,390,665 packets for this host	ips_delivered
330,882 packets for unknown/unsupported protocol	ips_noproto
97,939 packets forwarded	ips_forward
6,228 packets not forwardable	ips_cantforward
0 redirects sent	ips_redirectsent
29,447,726 packets sent from this host	ips_localout
769 packets sent with fabricated ip header	ips_rawout
0 output packets dropped due to no bufs, etc.	ips_odropped
0 output packets discarded due to no route	ips_noroute
260,484 output datagrams fragmented	ips_fragmented
796,084 fragments created	ips_ofragments
0 datagrams that can't be fragmented	ips_cantfrag

The value for ips\_noproto is high because it can count ICMP host unreachable messages when there is no process ready to receive the messages. See [Section 32.5](#) for more details.

## SNMP Variables

[Figure 8.6](#) shows the relationship between the SNMP variables in the IP group and the statistics collected by Net/3.

## Figure 8.6. Simple SNMP variables in IP

# group.

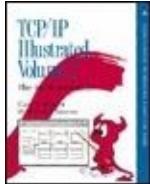
SNMP variable	ipstat member	Description
ipDefaultTTL	ip_defttl	default TTL for datagrams (64 "hops")
ipForwarding	ipforwarding	is system acting as a router?
ipReasmTimeout	IPFRAGTTL	reasembly timeout for fragments (30 seconds)
ipInReceives	ips_total	total #IP packets received
ipInHdrErrors	ips_badsum + ips_tooshort + ips_toosmall + ips_badhlen + ips_badlen + ips_badoptions + ips_badvers	#packets with errors in IP header
ipInAddrErrors	ips_cantforward	#IP packets discarded because of misdelivery (ip_output failure also)
ipForwDatagrams	ips_forward	#IP packets forwarded
ipReasmReqds	ips_fragments	#fragments received
ipReasmFails	ips_fragdropped + ips_fragtimeout	#fragments dropped
ipReasmOKs	ips_reassembled	#datagrams successfully reassembled
ipInDiscards	(not implemented)	#datagrams discarded because of resource limitations
ipInUnknownProtos	ips_noproto	#datagrams with an unknown or unsupported protocol
ipInDelivers	ips_delivered	#datagrams delivered to transport layer
ipOutRequests	ips_localout	#datagrams generated by transport layers
ipFragOKs	ips_fragmented	#datagrams successfully fragmented
ipFragFails	ips_cantfrag	#IP packets discarded because of don't fragment bit
ipFragCreates	ips_ofragments	#fragments created for output
ipOutDiscards	ips_odropped	#IP packets dropped because of resource shortages
ipOutNoRoutes	ips_noroute	#IP packets discarded because of no route

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



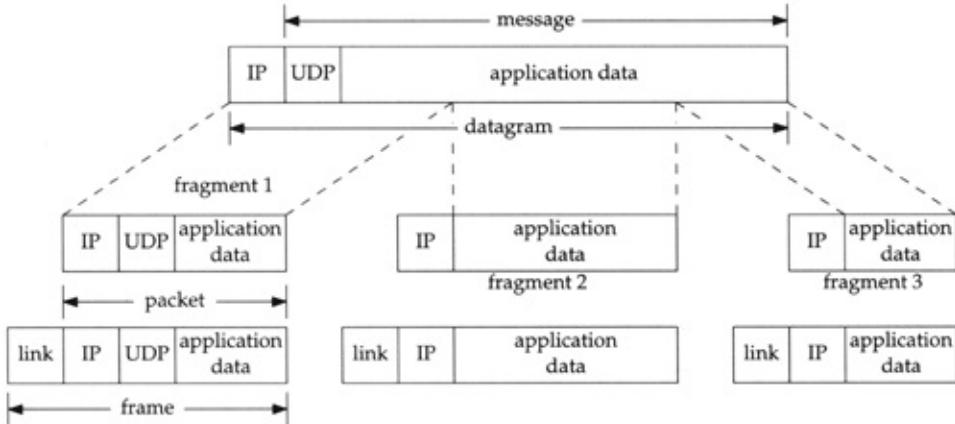
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.3 IP Packets

To be accurate while discussing Internet protocol processing, we must define a few terms. [Figure 8.7](#) illustrates the terms that describe data as it passes through the various Internet layers.

**Figure 8.7. Frames, packets, fragments, datagrams, and messages.**

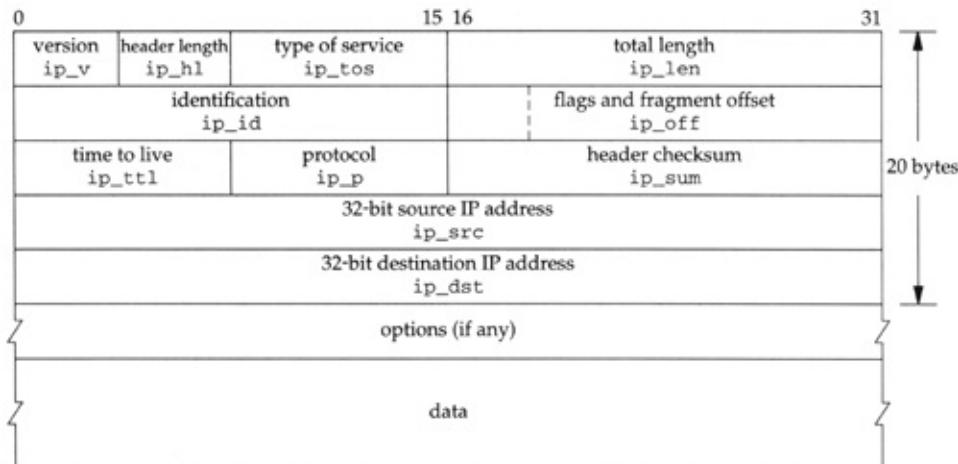


We call the data passed to IP by a transport protocol a *message*. A message typically contains a transport header and application data. UDP is the transport protocol illustrated in [Figure 8.7](#). IP prepends its own header to the message to form a *datagram*. If the datagram is too large for transmission on the selected network, IP splits the datagram into several *fragments*, each of which contains its own IP header and a portion of the original datagram. [Figure 8.7](#) shows a datagram split into three fragments.

An IP fragment or an IP datagram small enough to not require fragmentation are called *packets* when presented to the data-link layer for transmission. The data-link layer prepends its own header and transmits the resulting *frame*.

IP concerns itself only with the IP header and does not examine or modify the message itself (other than to perform fragmentation). [Figure 8.8](#) shows the structure of the IP header.

**Figure 8.8. IP datagram, including the ip structure names.**



[Figure 8.8](#) includes the member names of the ip structure (shown in [Figure 8.9](#)) through which Net/3 accesses the IP header.

**Figure 8.9. ip structure.**

---

```

40 /*
41 * Structure of an internet header, naked of options.
42 *
43 * We declare ip_len and ip_off to be short, rather than u_short
44 * pragmatically since otherwise unsigned comparisons can result
45 * against negative integers quite easily, and fail in subtle ways.
46 */
47 struct ip {
48 #if BYTE_ORDER == LITTLE_ENDIAN
49     u_char    ip_hl:4;           /* header length */
50             ip_v:4;           /* version */
51 #endif
52 #if BYTE_ORDER == BIG_ENDIAN
53     u_char    ip_v:4;           /* version */
54             ip_hl:4;           /* header length */
55 #endif
56     u_char    ip_tos;          /* type of service */
57     short     ip_len;          /* total length */
58     u_short   ip_id;          /* identification */
59     short     ip_off;          /* fragment offset field */
60 #define IP_DF 0x4000          /* dont fragment flag */
61 #define IP_MF 0x2000          /* more fragments flag */
62 #define IP_OFMASK 0x1fff       /* mask for fragmenting bits */
63     u_char    ip_ttl;          /* time to live */
64     u_char    ip_P;           /* protocol */
65     u_short   ip_sum;          /* checksum */
66     struct in_addr ip_src, ip_dst; /* source and dest address */
67 };

```

---

ip.h

## 47-67

Since the physical order of bit fields in memory is machine and compiler dependent, the #ifs ensure that the compiler lays out the structure members in the order specified by the IP standard. In this way, when Net/3 overlays an ip structure on an IP packet in memory, the structure members access the correct bits in the packet.

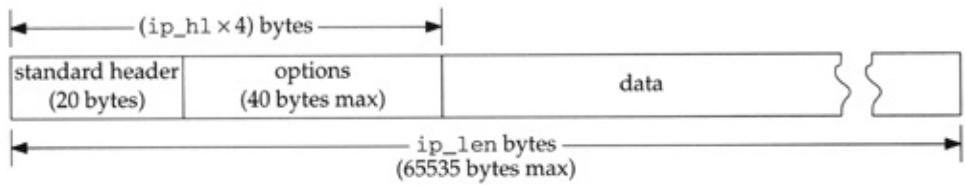
The IP header contains the format of the IP packet and its contents along with addressing, routing, and fragmentation

information.

The format of an IP packet is specified by `ip_v`, the version, which is always 4; `ip_hl`, the header length measured in 4-byte units; `ip_len`, the packet length measured in bytes; `ip_p`, the transport protocol that created the data within the packet; and `ip_sum`, the checksum that detects changes to the header while in transit.

A standard IP header is 20 bytes long, so `ip_hl` must be greater than or equal to 5. A value greater than 5 indicates that IP options appear just after the standard header. The maximum value of `ip_hl` is 15 ( $2^4 - 1$ ), which allows for up to 40 bytes of options ( $20 + 40 = 60$ ). The maximum length of an IP datagram is 65535 ( $2^{16} - 1$ ) bytes since `ip_len` is a 16-bit field. [Figure 8.10](#) illustrates this organization.

**Figure 8.10. Organization of an IP packet with options.**



Because ip\_hl is measured in 4-byte units, IP options must always be padded to a 4-byte boundary.

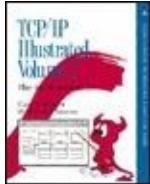
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.4 Input Processing: ipintr Function

In [Chapters 3, 4, and 5](#) we described how our example network interfaces queue incoming datagrams for protocol processing:

#### **1. The Ethernet interface demultiplexes incoming frames with the type field found in the Ethernet header ([Section 4.3](#)).**

- The SLIP interface handles only IP packets, so demultiplexing is unnecessary ([Section 5.3](#)).
- The loopback interface combines output

and input processing in the function looutput and demultiplexes datagrams with the `sa_family` member of the destination address ([Section 5.4](#)).

In each case, after the interface queues the packet on `ipintrq`, it schedules a software interrupt through `schednetisr`. When the software interrupt occurs, the kernel calls `ipintr` if IP processing has been scheduled by `schednetisr`. Before the call to `ipintr`, the CPU priority is changed to `splnet`.

## ipintr Overview

`ipintr` is a large function that we discuss in four parts: (1) verification of incoming packets, (2) option processing and forwarding, (3) packet reassembly, and (4).demultiplexing. Packet reassembly occurs in `ipintr`, but it is complex enough that we discuss it separately in [Chapter 10](#). [Figure 8.11](#) shows the overall organization of `ipintr`.

## Figure 8.11. ipintr function.

```
ip_input.c
100 void
101 ipintr()
102 {
103     struct ip *ip;
104     struct mbuf *m;
105     struct ipq *fp;
106     struct in_ifaddr *ia;
107     int     hlen, s;
108
109     next:
110     /*
111      * Get next datagram off input queue and get IP header
112      * in first mbuf.
113      */
114     s = splimp();
115     IF_DEQUEUE(&ipintrq, m);
116     splx(s);
117     if (m == 0)
118         return;
119
120     /* input packet processing */
121     /* Figures 8.12, 8.13, 8.15, 10.11, and 12.40 */
122
123     goto next;
124 bad:
125     m_freem(m);
126     goto next;
127 }
```

ip\_input.c

100-117

The label next marks the start of the main packet processing loop. ipintr removes packets from ipintrq and processes them until the queue is empty. If control falls through to the end of the function, the goto passes control back to the top of the function at next. ipintr blocks incoming packets with splimp so that the network interrupt routines (such as slinput and ether\_input) don't run while it accesses

the queue.

332-336

The label bad marks the code that silently discards packets by freeing the associated mbuf and returning to the top of the processing loop at next. Throughout ipintr, errors are handled by jumping to bad.

## Verification

We start with [Figure 8.12](#): dequeuing packets from ipintrq and verifying their contents. Damaged or erroneous packets are silently discarded.

**Figure 8.12. ipintr function.**

---

```

118     /*
119      * If no IP addresses have been set yet but the interfaces
120      * are receiving, can't do anything with incoming packets yet.
121      */
122     if (in_ifaddr == NULL)
123         goto bad;
124     ipstat.ips_total++;
125     if (m->m_len < sizeof(struct ip) &&
126         (m = m_pullup(m, sizeof(struct ip))) == 0) {
127         ipstat.ips_toosmall++;
128         goto next;
129     }
130     ip = mtod(m, struct ip *);
131     if (ip->ip_v != IPVERSION) {
132         ipstat.ips_badvers++;
133         goto bad;
134     }
135     hlen = ip->ip_hl << 2;
136     if (hlen < sizeof(struct ip)) { /* minimum header length */
137         ipstat.ips_badhlen++;
138         goto bad;
139     }
140     if (hlen > m->m_len) {
141         if ((m = m_pullup(m, hlen)) == 0) {
142             ipstat.ips_badhlen++;
143             goto next;
144         }
145         ip = mtod(m, struct ip *);
146     }
147     if (ip->ip_sum = in_cksum(m, hlen)) {
148         ipstat.ips_badsum++;
149         goto bad;
150     }
151     /*
152      * Convert fields to host representation.
153      */
154     ntohs(ip->ip_len);
155     if (ip->ip_len < hlen) {
156         ipstat.ips_badlen++;
157         goto bad;
158     }
159     ntohs(ip->ip_id);
160     ntohs(ip->ip_off);

161     /*
162      * Check that the amount of data in the buffers
163      * is as at least much as the IP header would have us expect.
164      * Trim mbufs if longer than we expect.
165      * Drop packet if shorter than we expect.
166      */
167     if (m->m_pkthdr.len < ip->ip_len) {
168         ipstat.ips_tooshort++;
169         goto bad;
170     }

171     if (m->m_pkthdr.len > ip->ip_len) {
172         if (m->m_len == m->m_pkthdr.len) {
173             m->m_len = ip->ip_len;
174             m->m_pkthdr.len = ip->ip_len;
175         } else
176             m_adj(m, ip->ip_len - m->m_pkthdr.len);
177     }

```

---

ip\_input.c

## IP version

## 118-134

If the `in_ifaddr` list ([Section 6.5](#)) is empty, no IP addresses have been assigned to the network interfaces, and `ipintr` must discard all IP packets; without addresses, `ipintr` can't determine whether the packet is addressed to the system. Normally this is a transient condition occurring during system initialization when the interfaces are operating but have not yet been configured. We described address assignment in [Section 6.6](#).

Before `ipintr` accesses any IP header fields, it must verify that `ip_v` is 4 (IPVERSION). RFC 1122 requires an implementation to silently discard packets with unrecognized version numbers.

Net/2 didn't check `ip_v`. Most IP implementations in use today, including Net/2, were created after IP version 4 was standardized and have never needed to distinguish between packets from different IP versions. Since revisions to IP are now in progress, implementations in the near future will

have to check ip\_v.

IEN 119 [[Forgie 1979](#)] and RFC 1190 [[Topolcic 1990](#)] describe experimental protocols using IP versions 5 and 6. Version 6 has also been selected as the version for the next revision to the official IP standard (IPv6). Versions 0 and 15 are reserved, and the remaining versions are unassigned.

In C, the easiest way to process data located in an untyped area of memory is to overlay a structure on the area of memory and process the structure members instead of the raw bytes. As described in [Chapter 2](#), an mbuf chain stores a logical sequence of bytes, such as an IP packet, into many physical mbufs connected to each other on a linked list. Before the overlay technique can be applied to the IP packet headers, the header must reside in a contiguous area of memory (i.e., it isn't split between two mbufs).

135-146

The following steps ensure that the IP

header (including options) is in a contiguous area of memory:

- If the data within the first mbuf is smaller than a standard IP header (20 bytes), `m_pullup` relocates the standard header into a contiguous area of memory.

It is improbable that the link layer would split even the largest (60 bytes) IP header into two mbufs necessitating the use of `m_pullup` as described.

- `ip_hl` is multiplied by 4 to get the header length in bytes, which is saved in `hlen`.
- If `hlen`, the length of the IP packet header in bytes, is less than the length of a standard header (20 bytes), it is invalid and the packet is discarded.
- If the entire header is still not in the first mbuf (i.e., the packet contains IP options), `m_pullup` finishes the job.

Again, this should not be necessary.

Checksum processing is an important part of all the Internet protocols. Each protocol uses the same algorithm (implemented by the function `in_cksum`) but on different parts of the packet. For IP, the checksum protects only the IP header (and options if present). For transport protocols, such as UDP or TCP, the checksum covers the data portion of the packet and the transport header.

## IP checksum

147-150

`ipintr` stores the checksum computed by `in_cksum` in the `ip_sum` field of the header. An undamaged header should have a checksum of 0.

As we'll see in [Section 8.7](#), `ip_sum` must be cleared before the checksum on an outgoing packet is computed. By storing the result from `in_cksum` in `ip_sum`, the packet is prepared for forwarding (although the TTL has not been decremented yet). The `ip_output`

function does not depend on this behavior; it recomputes the checksum for the forwarded packet.

If the result is nonzero the packet is silently discarded. We discuss `in_cksum` in more detail in [Section 8.7](#).

## Byte ordering

### 151-160

The Internet standards are careful to specify the byte ordering of multibyte integer values in protocol headers. `NTOHS` converts all the 16-bit values in the IP header from network byte order to host byte order: the packet length (`ip_len`), the datagram identifier (`ip_id`), and the fragment offset (`ip_off`). `NTOHS` is a null macro if the two formats are the same. Conversion to host byte order here obviates the need to perform a conversion every time Net/3 examines the fields.

## Packet length

## 161-177

If the logical size of the packet (`ip_len`) is greater than the amount of data stored in the mbuf chain (`m_pkthdr.len`), some bytes are missing and the packet is dropped. If the mbuf chain is larger than the packet, the extra bytes are trimmed.

A common cause for lost bytes is data arriving on a serial device with little or no buffering, such as on many personal computers. The incoming bytes are discarded by the device and IP discards the resulting packet.

These extra bytes may arise, for example, on an Ethernet device when an IP packet is smaller than the minimum size required by Ethernet. The frame is transmitted with extra bytes that are discarded here. This is one reason why the length of the IP packet is stored in the header; IP allows the link layer to pad packets.

At this point, the complete IP header is available, the logical size and the physical

size of the packet are the same, and the checksum indicates that the header arrived undamaged.

## To Forward or Not To Forward?

The next section of ipintr, shown in [Figure 8.13](#), calls ip\_dooptions ([Chapter 9](#)) to process IP options and then determines whether or not the packet has reached its final destination. If it hasn't reached its final destination, Net/3 may attempt to forward the packet (if the system is configured as a router). If it has reached its final destination, it is passed to the appropriate transport-level protocol.

**Figure 8.13. ipintr continued.**

```

178  /*
179   * Process options and, if not destined for us,
180   * ship it on. ip_dooptions returns 1 when an
181   * error was detected (causing an icmp message
182   * to be sent and the original packet to be freed).
183   */
184  ip_nhops = 0;           /* for source routed packets */
185  if (hlen > sizeof(struct ip) && ip_dooptions(m))
186      goto next;
187  /*
188   * Check our list of addresses, to see if the packet is for us.
189   */
190  for (ia = in_ifaddr; ia; ia = ia->ia_next) {
191 #define satosin(sa) ((struct sockaddr_in *) (sa))
192      if (IA_SIN(ia)->sin_addr.s_addr == ip->ip_dst.s_addr)
193          goto ours;
194      /* Only examine broadcast addresses for the receiving interface */
195      if (ia->ia_ifp == m->m_pkthdr.rcvif &&
196          (ia->ia_ifp->if_flags & IFF_BROADCAST)) {
197          u_long t;
198          if (satosin(&ia->ia_broadaddr)->sin_addr.s_addr ==
199              ip->ip_dst.s_addr)
200              goto ours;
201          if (ip->ip_dst.s_addr == ia->ia_netbroadcast.s_addr)
202              goto ours;
203          /*
204           * Look for all-0's host part (old broadcast addr),
205           * either for subnet or net.
206           */
207          t = ntohl(ip->ip_dst.s_addr);
208          if (t == ia->ia_subnet)
209              goto ours;
210          if (t == ia->ia_net)
211              goto ours;
212      }
213  }

    /* multicast code (Figure 12.39) */

```

```

258  if (ip->ip_dst.s_addr == (u_long) INADDR_BROADCAST)
259      goto ours;
260  if (ip->ip_dst.s_addr == INADDR_ANY)
261      goto ours;
262  /*
263   * Not for us; forward if possible and desirable.
264   */
265  if (ipforwarding == 0) {
266      ipstat.ips_cantforward++;
267      m_free(m);
268  } else
269      ip_forward(m, 0);
270  goto next;
271 ours:

```

## Option processing

178-186

The source route from the previous packet is discarded by clearing ip\_nhops ([Section 9.6](#)). If the packet header is larger than a default header, it must include options that are processed by ip\_dooptions. If ip\_dooptions returns 0, ipintr should continue processing the packet; otherwise ip\_dooptions has completed processing of the packet by forwarding or discarding it, and ipintr can process the next packet on the input queue. We postpone further discussion of option processing until [Chapter 9](#).

After option processing, ipintr decides whether the packet has reached its final destination by comparing ip\_dst in the IP header with the IP addresses configured for all the local interfaces, ipintr must consider several broadcast addresses, one or more unicast addresses, and any multicast addresses that are associated with the interface.

## Final destination?

187-261

ipintr starts by traversing in\_ifaddr ([Figure 6.5](#)), the list of configured Internet addresses, to see if there is a match with the destination address of the packet. A series of comparisons are made for each in\_ifaddr structure found in the list. There are four general cases to consider:

- an exact match with one of the interface addresses (first row of [Figure 8.14](#)),

**Figure 8.14. Comparisons to determine whether or not a packet has reached its final destination.**

Variable	Ethernet	SLIP	Loopback	Lines (Figure 8.13)
ia_addr	140.252.13.33	140.252.1.29	127.0.0.1	192–193
ia_broadaddr	140.252.13.63			198–200
ia_netbroadcast	140.252.255.255			201–202
ia_subnet	140.252.13.32			207–209
ia_net	140.252.0.0			210–211
INADDR_BROADCAST		255.255.255.255		258–259
INADDR_ANY		0.0.0.0		260–261

- a match with the one of the broadcast addresses associated with the *receiving* interface (middle four rows of [Figure 8.14](#)),
- a match with one of the multicast groups associated with the *receiving* interface ([Figure 12.39](#)), or
- a match with one of the two limited broadcast addresses (last row of [Figure 8.14](#)).

[Figure 8.14](#) illustrates the addresses that would be tested for a packet arriving on the Ethernet interface of the host sun in our sample network, excluding multicast

addresses, which we discuss in [Chapter 12](#).

The tests with ia\_subnet, ia\_net, and INADDR\_ANY are not required as they represent obsolete broadcast addresses used by 4.2BSD. Unfortunately, many TCP/IP implementations have been derived from 4.2BSD, so it may be important to recognize these old broadcast addresses on some networks.

## Forwarding

262-271

If ip\_dst does not match any of the addresses, the packet has not reached its final destination. If ipforwarding is not set, the packet is discarded. Otherwise, ip\_forward attempts to route the packet toward its final destination.

A host may discard packets that arrive on an interface other than the one specified by the destination address of the packet. In this case, Net/3 would

not search the entire `in_ifaddr` list; only addresses assigned to the receiving interface would be considered. RFC 1122 calls this a *strong end system* model.

For a multihomed host, it is uncommon for a packet to arrive at an interface that does not correspond to the packet's destination address, unless specific host routes have been configured. The host routes force neighboring routers to consider the multihomed host as the next-hop router for the packets. The *weak end system* model requires that the host accept these packets. An implementor is free to choose either model. Net/3 implements the weak end system model.

## Reassembly and Demultiplexing

Finally, we look at the last section of `ipintr` ([Figure 8.15](#)) where reassembly and demultiplexing occur. We have omitted the reassembly code and postpone its

discussion until [Chapter 10](#). The omitted code sets the pointer ip to null if it could not reassemble a complete datagram. Otherwise, ip points to a complete datagram that has reached its final destination.

## Figure 8.15. ipintr continued.

```
----- ip_input.c
      /* reassembly (Figure 10.11) */

325  /*
326   * If control reaches here, ip points to a complete datagram.
327   * Otherwise, the reassembly code jumps back to next (Figure 8.11)
328   * Switch out to protocol's input routine.
329   */
330 ipstat.ips_delivered++;
331 (*inetsw[ip_protox[ip->ip_p]].pr_input) (m, hlen);
332 goto next;
----- ip_input.c
```

## Transport demultiplexing

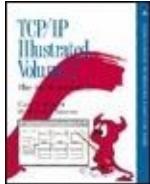
325-332

The protocol specified in the datagram (ip\_p) is mapped with the ip\_protox array ([Figure 7.22](#)) to an index into the inetsw array, ipintr calls the pr\_input function from the selected protosw structure to process the transport message contained

within the datagram. When pr\_input returns, ipintr proceeds with the next packet on ipintrq.

It is important to notice that transport-level processing for each packet occurs within the processing loop of ipintr. There is no queueing of incoming packets between IP and the transport protocols, unlike the queueing in SVR4 streams implementations of TCP/IP.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.5 Forwarding: `ip_forward` Function

A packet arriving at a system other than its final destination needs to be forwarded. `ipintr` calls the function `ip_forward`, which implements the forwarding algorithm, only when `ipforwarding` is nonzero ([Section 6.1](#)) or when the packet includes a source route ([Section 9.6](#)). When the packet includes a source route, `ip_dooptions` calls `ip_forward` with the second argument, `srcrt`, set to 1.

`ip_forward` interfaces with the routing tables through a route structure shown in [Figure 8.16](#)

## Figure 8.16. route structure.

```
46 struct route {  
47     struct rtentry *ro_rt;      /* pointer to struct with information */  
48     struct sockaddr ro_dst;    /* destination of this route */  
49 };
```

-route.h

-route.h

46-49

There are only two members in a route structure: ro\_rt, a pointer to an rtentry structure; and ro\_dst, a sockaddr structure, which specifies the destination associated with the route entry pointed to by ro\_rt. The destination is the key used to find route information in the kernel's routing tables. [Chapter 18](#) has a detailed description of the rtentry structure and the routing tables.

We show ip\_forward in two parts. The first part makes sure the system is permitted to forward the packet, updates the IP header, and selects a route for the packet. The second part handles ICMP redirect messages and passes the packet to ip\_output for transmission.

## Is packet eligible for forwarding?

867-871

The first argument to `ip_forward` is a pointer to an mbuf chain containing the packet to be forwarded. If the second argument, `srcrt`, is nonzero, the packet is being forwarded because of a source route option ([Section 9.6](#)).

879-884

The if statement identifies and discards the following packets:

- link-level broadcasts

Any network interface driver that supports broadcasts must set the `M_BCAST` flag for a packet received as a broadcast. `ether_input` ([Figure 4.13](#)) sets `M_BCAST` if the packet was addressed to the Ethernet broadcast address. Link-level broadcast packets are never forwarded.

Packets addressed to a unicast IP address but sent as a link-level broadcast are prohibited by RFC 1122 and are discarded here.

- loopback packets

`in_canforward` returns 0 for packets addressed to the loopback network.

These packets may have been passed to `ip_forward` by `ipintr` because the loopback interface was not configured correctly.

- network 0 and class E addresses

`in_canforward` returns 0 for these packets. These destination addresses are invalid and packets addressed to them should not be circulating in the network since no host will accept them.

- class D addresses

Packets addressed to a class D address should be processed by the multicast forwarding function, `ip_mforward`, not by `ip_forward`. `in_canforward` rejects class D (multicast) addresses.

RFC 791 specifies that every system that processes a packet must decrement the time-to-live (TTL) field by at least 1 even though TTL is measured in seconds.

Because of this requirement, TTL is usually considered a bound on the number of hops an IP packet may traverse before being discarded. Technically, a router that held a packet for more than 1 second could decrement ip\_ttl by more than 1.

**Figure 8.17. ip\_forward function: route selection.**

---

```

867 void
868 ip_forward(m, srcrt)
869 struct mbuf *m;
870 int      srcrt;
871 {
872     struct ip *ip = mtod(m, struct ip *);
873     struct sockaddr_in *sin;
874     struct rtentry *rt;
875     int      error, type = 0, code;
876     struct mbuf *mcopy;
877     n_long dest;
878     struct ifnet *destifp;
879     dest = 0;
880     if (m->m_flags & M_BCAST || in_canforward(ip->ip_dst) == 0) {
881         ipstat.ips_cantforward++;
882         m_free(m);
883         return;
884     }
885     HTONS(ip->ip_id);
886     if (ip->ip_ttl <= IPTTLDEC) {
887         icmp_error(m, ICMP_TIMXCEED, ICMP_TIMXCEED_INTRANS, dest, 0);
888         return;
889     }
890     ip->ip_ttl -= IPTTLDEC;
891     sin = (struct sockaddr_in *) &ipforward_rt.ro_dst;
892     if ((rt = ipforward_rt.ro_rt) == 0 ||
893         ip->ip_dst.s_addr != sin->sin_addr.s_addr) {
894         if (ipforward_rt.ro_rt) {
895             RTPFREE(ipforward_rt.ro_rt);
896             ipforward_rt.ro_rt = 0;
897         }
898         sin->sin_family = AF_INET;
899         sin->sin_len = sizeof(*sin);
900         sin->sin_addr = ip->ip_dst;
901         rtalloc(&ipforward_rt);
902         if (ipforward_rt.ro_rt == 0) {
903             icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_HOST, dest, 0);
904             return;
905         }
906         rt = ipforward_rt.ro_rt;
907     }
908     /*
909      * Save at most 64 bytes of the packet in case
910      * we need to generate an ICMP message to the src.
911      */
912     mcopy = m_copy(m, 0, imin((int) ip->ip_len, 64));
913     ip_ifmatrix[rt->rt_ifp->if_index +
914                 if_index * m->m_pkthdr.rcvif->if_index]++;

```

---

ip\_input.c

The question arises: How long is the longest path in the Internet? This metric is called the *diameter* of a network. There is no way to discover the diameter other than through empirical methods. A 37-hop path was posted in [Olivier 1994].

## Decrement TTL

885-890

The packet identifier is converted back to network byte order since it isn't needed for forwarding and it should be in the correct order if ip\_forward sends an ICMP error message, which includes the invalid IP header.

Net/3 neglects to convert ip\_len, which ipintr converted to host byte order. The authors noted that on big endian machines this does not cause a problem since the bytes are never swapped. On little endian machines, such as a 386, this bug allows the byte-swapped value to be returned in the IP header within the ICMP error. This bug was observed in ICMP packets returned from SVR4 (probably Net/1 code) running on a 386 and from AIX 3.2 (4.3BSD Reno code).

If ip\_ttl has reached 1 (IPTTLDEC), an ICMP time exceeded message is returned to the sender and the packet is discarded. Otherwise, ip\_forward decrements ip\_ttl

by IPTTLDEC.

A system should never receive an IP datagram with a TTL of 0, but Net/3 generates the correct ICMP error if this happens since ip\_ttl is examined after the packet is considered for local delivery and before it is forwarded.

## Locate next hop

891-907

The IP forwarding algorithm caches the most recent route, in the global route structure ipforward\_rt, and applies it to the current packet if possible. Research has shown that consecutive packets tend to have the same destination address ([[Jain and Routhier 1986](#)] and [[Mogul 1991](#)]), so this *one-behind* cache minimizes the number of routing lookups. If the cache (ipforward\_rt) is empty or the current packet is to a different destination than the route entry in ipforward\_rt, the previous route is discarded, ro\_dst is initialized to the new destination, and

rtalloc finds a route to the current packet's destination. If no route can be found for the destination, an ICMP host unreachable error is returned and the packet discarded.

## 908-914

Since ip\_output discards the packet when an error occurs, m\_copy makes a copy of the first 64 bytes in case ip\_forward sends an ICMP error message. ip\_forward does not abort if the call to m\_copy fails. In this case, the error message is not sent.

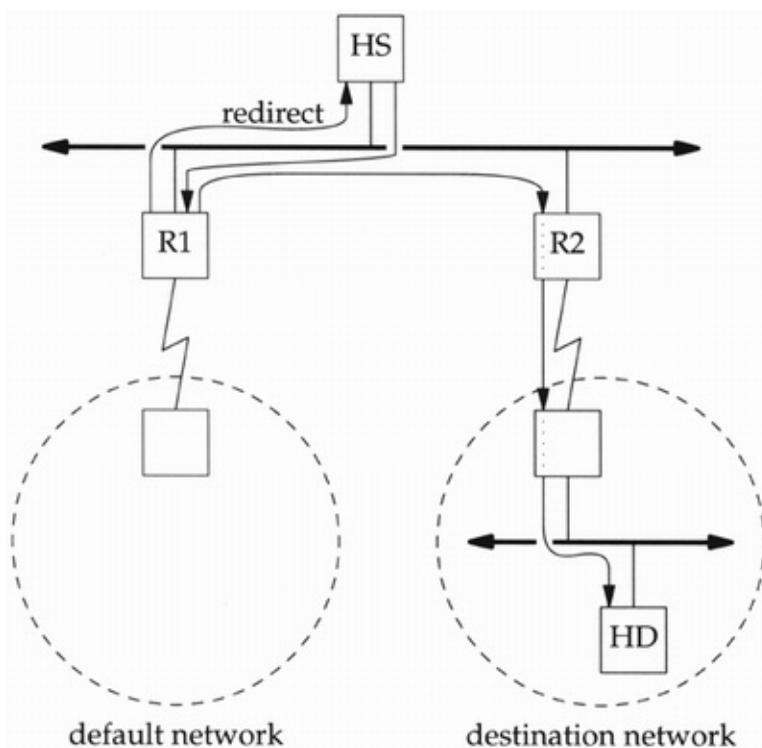
ip\_ifmatrix records the number of packets routed between interfaces. The counter with the indexes of the receiving and sending interfaces is incremented.

## Redirect Messages

A first-hop router returns an ICMP redirect message to the source host when the host incorrectly selects the router as the packet's first-hop destination. The IP networking model assumes that hosts are relatively ignorant of the overall internet

topology and assigns the responsibility of maintaining correct routing tables to routers. A redirect message from a router informs a host that it has selected an incorrect route for a packet. We use [Figure 8.18](#) to illustrate redirect messages.

**Figure 8.18. Router R1 is redirecting host HS to use router R2 to reach HD.**



Generally, an administrator configures a host to send packets for remote networks to a default router. In [Figure 8.18](#), host HS

has R1 configured as its default router. When it first attempts to send a packet to HD it sends the packet to R1, not knowing that R2 is the appropriate choice. R1 recognizes the mistake, forwards the packet to R2, and sends a redirect message back to HS. After receiving the redirect, HS updates its routing tables so that the next packet to HD is sent directly to R2.

RFC 1122 recommends that only routers send redirect messages and that hosts must update their routing tables when receiving ICMP redirect messages ([Section 11.8](#)). Since Net/3 calls `ip_forward` only when the system is configured as a router, Net/3 follows RFC 1122's recommendations.

In [Figure 8.19](#), `ip_forward` determines whether or not it should send a redirect message.

**Figure 8.19. `ip_forward` continued.**

```

915  /*
916   * If forwarding packet is using same interface that it came in on,
917   * perhaps should send a redirect to sender to shortcut a hop.
918   * Only send redirect if source is sending directly to us,
919   * and if packet was not source routed (or has any options).
920   * Also, don't send redirect if forwarding using a default route
921   * or a route modified by a redirect.
922   */
923 #define satosin(sa) ((struct sockaddr_in *)(sa))
924     if (rt->rt_ifp == m->m_pkthdr.rcvif &&
925         (rt->rt_flags & (RTF_DYNAMIC | RTF_MODIFIED)) == 0 &&
926         satosin(rt_key(rt))->sin_addr.s_addr != 0 &&
927         ipsendredirects && !srcrt) {
928 #define RTA(rt) ((struct in_ifaddr *)(rt->rt_ifa))
929     u_long src = ntohl(ip->ip_src.s_addr);

930     if (RTA(rt) &&
931         (src & RTA(rt)->ia_subnetmask) == RTA(rt)->ia_subnet) {
932         if (rt->rt_flags & RTF_GATEWAY)
933             dest = satosin(rt->rt_gateway)->sin_addr.s_addr;
934         else
935             dest = ip->ip_dst.s_addr;
936         /* Router requirements says to only send host redirects */
937         type = ICMP_REDIRECT;
938         code = ICMP_REDIRECT_HOST;
939     }
940 }

```

ip\_input.c

## Leaving on receiving interface?

915-929

The rules by which a router recognizes redirect situations are complicated. First, redirects are applicable only when a packet is received and resent on the same interface (`rt_ifp` and `rcvif`). Next, the selected route must not have been itself created or modified by an ICMP redirect message (`RTF_DYNAMIC` | `RTF_MODIFIED`), nor can the route be to the default destination (0.0.0.0). This ensures that the system does not

propagate routing information for which it is not an authoritative source, and that it does not share its default route with other systems.

Generally, routing protocols use the special destination 0.0.0.0 to locate a default route. When a specific route to a destination is not available, the route associated with destination 0.0.0.0 directs the packet toward a default router.

[Chapter 18](#) has more information about default routes.

The global integer `ipsendredirects` specifies whether the system has administrative authority to send redirects ([Section 8.9](#)). By default, `ipsendredirects` is 1. Redirects are suppressed when the system is source routing a packet as indicated by the `srcrt` argument passed to `ip_forward`, since presumably the source host wanted to override the decisions of the intermediate routers.

## Send redirect?

## 930-931

This test determines if the packet originated on the local subnet. If the subnet mask bits of the source address and the outgoing interface's address are the same, the addresses are on the same IP network. If the source and the outgoing interface are on the same network, then this system should not have received the packet, since the source could have sent the packet directly to the correct first-hop router. The ICMP redirect message informs the host of the correct first-hop destination. If the packet originated on some other subnet, then the previous system was a router and this system does not send a redirect; the mistake will be corrected by a routing protocol.

In any case, routers are required to ignore redirect messages. Despite the requirement, Net/3 does not discard redirect messages when ipforwarding is set (i.e., when it is configured to be a router).

## Select appropriate router

932-940

The ICMP redirect message contains the address of the correct next system, which is a router's address if the destination host is not on the directly connected network or the host address if the destination host is on the directly connected network.

RFC 792 describes four types of redirect messages: (1) network, (2) host, (3) TOS and network, and (4) TOS and host. RFC 1009 recommends against sending network redirects at any time because of the impossibility of guaranteeing that the host receiving the redirect can determine the appropriate subnet mask for the destination network. RFC 1122 recommends that hosts treat network redirects as host redirects to avoid this ambiguity. Net/3 sends only host redirects and ignores any TOS considerations. In [Figure 8.20](#), ipintr passes the packet and any ICMP messages to the link layer.

**Figure 8.20. ip\_forward continued.**

```

941     error = ip_output(m, (struct mbuf *) 0, &ipforward_rt,
942                         IP_FORWARDING | IP_ALLOWBROADCAST, 0);
943     if (error)
944         ipstat.ips_cantforward++;
945     else {
946         ipstat.ips_forward++;
947         if (type)
948             ipstat.ips_redirectsent++;
949         else {
950             if (mcopy)
951                 m_freem(mcopy);
952             return;
953         }
954     }
955     if (mcopy == NULL)
956         return;
957     destifp = NULL;
958
959     switch (error) {
960
961     case 0:           /* forwarded, but need redirect */
962         /* type, code set above */
963         break;
964
965     case ENETUNREACH:      /* shouldn't happen, checked above */
966     case EHOSTUNREACH:
967     case ENETDOWN:
968     case EHOSTDOWN:
969     default:
970         type = ICMP_UNREACH;
971         code = ICMP_UNREACH_HOST;
972         break;
973
974     case EMSGSIZE:
975         type = ICMP_UNREACH;
976         code = ICMP_UNREACH_NEEDFRAG;
977         if (ipforward_rt.ro_rt)
978             destifp = ipforward_rt.ro_rt->rt_ifp;
979         ipstat.ips_cantfrag++;
980         break;
981
982     case ENOBUFS:
983         type = ICMP_SOURCEQUENCH;
984         code = 0;
985         break;
986     }
987     icmp_error(mcopy, type, code, dest, destifp);
988 }

```

---

*ip\_input.c*

The redirect messages were standardized before subnetting. In a nonsubnetted internet, network redirects are useful but in a subnetted internet they are ambiguous since they do not include a subnet mask.

## Forward packet

941-954

At this point, ip\_forward has a route for the packet and has determined if an ICMP redirect is warranted. ip\_output sends the packet to the next hop as specified in the route ipforward\_rt. The IP\_ALLOWBROADCAST flag allows the packet being forwarded to be a directed broadcast to a local network. If ip\_output succeeds and no redirect message needs to be sent, the copy of the first 64 bytes of the packet is discarded and ip\_forward returns.

## Send ICMP error?

955-983

ip\_forward may need to send an ICMP message because ip\_output failed or a redirect is pending. If there is no copy of the original packet (there might have been a buffer shortage at the time the copy was attempted), the message can't be sent and

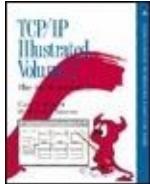
`ip_forward` returns. If a redirect is pending, type and code have been previously set, but if `ip_output` failed, the switch statement sets up the new ICMP type and code values based on the return value from `ip_output`. `icmp_error` sends the message. The ICMP message from a failed `ip_output` overrides any pending redirect message.

It is important to recognize the significance of the switch statement that handles errors from `ip_output`. It translates local system errors into the appropriate ICMP error message, which is returned to the packet's source. [Figure 8.21](#) summarizes the errors. Chapter 11 describes the ICMP messages in more detail.

## Figure 8.21. Errors from `ip_output`.

Error code from ip_output	ICMP message generated	Description
EMSGSIZE	ICMP_UNREACH_NEEDFRAG	The outgoing packet was too large for the selected interface and fragmentation was prohibited (Chapter 10).
ENOBUFS	ICMP_SOURCEQUENCH	The interface queue is full or the kernel is running short of free memory. This message is an indication to the source host to lower the data rate.
EHOSTUNREACH ENETDOWN		A route to the host could not be found. The outgoing interface specified by the route is not operating.
EHOSTDOWN	ICMP_UNREACH_HOST	The interface could not send the packet to the selected host.
default		Any unrecognized error is reported as an ICMP_UNREACH_HOST error.

Net/3 always generates the ICMP source quench when ip\_output returns ENOBUFS. The Router Requirements RFC [[Almquist and Kastenholz 1994](#)] deprecate the source quench and state that a router should not generate them.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.6 Output Processing: `ip_output` Function

The IP output code receives packets from two sources: `ip_forward` and the transport protocols ([Figure 8.1](#)). It would seem reasonable to expect IP output operations to be accessed by `inetsw[0].pr_output`, but this is not the case. The standard Internet transport protocols (ICMP, IGMP, UDP, and TCP) call `ip_output` directly instead of going through the `inetsw` table. For the standard Internet transport protocols, the generality of the `protosw` structure is not necessary, since the calling functions are not accessing IP in a protocol-independent context. In [Chapter 20](#) we'll see that the protocol-independent routing sockets call

`pr_output` to access IP.

We describe `ip_output` in three sections:

- header initialization,
- route selection, and
- source address selection and fragmentation.

## Header Initialization

The first section of `ip_output`, shown in [Figure 8.22](#), merges options into the outgoing packet and completes the IP header for packets that are passed from the transport protocols (not those from `ip_forward`).

**Figure 8.22. `ip_output` function.**

```
44 int  
45 ip_output(m0, opt, ro, flags, imo)  
46 struct mbuf *m0;  
47 struct mbuf *opt;  
48 struct route *ro;  
49 int     flags;  
50 struct ip_moptions *imo;  
51 {  
52     struct ip *ip, *mhip;  
53     struct ifnet *ifp;  
54     struct mbuf *m = m0;  
55     int      hlen = sizeof(struct ip);  
56     int      len, off, error = 0;  
57     struct route iproute;  
58     struct sockaddr_in *dst;  
59     struct in_ifaddr *ia;  
60     if (opt) {  
61         m = ip_insertoptions(m, opt, &len);  
62         hlen = len;  
63     }  
64     ip = mtod(m, struct ip *);  
65     /*  
66      * Fill in IP header.  
67      */  
68     if ((flags & (IP_FORWARDING | IP_RAWOUTPUT)) == 0) {  
69         ip->ip_v = IPVERSION;  
70         ip->ip_off &= IP_DF;  
71         ip->ip_id = htons(ip_id++);  
72         ip->ip_hl = hlen >> 2;  
73         ipstat.ips_localout++;  
74     } else {  
75         hlen = ip->ip_hl << 2;  
76     }
```

ip\_output.c

## 44-59

The arguments to ip\_output are: m0, the packet to send; opt, the IP options to include; ro, a cached route to the destination; flags, described in [Figure 8.23](#); and imo, a pointer to multicast options described in [Chapter 12](#).

## Figure 8.23. ip\_output:flags values.

Flag	Description
<i>IP_FORWARDING</i>	This is a forwarded packet.
<i>IP_ROUTETOIF</i>	Ignore routing tables and route directly to interface.
<i>IP_ALLOWBROADCAST</i>	Allow broadcast packets to be sent.
<i>IP_RAWOUTPUT</i>	Packet contains a preconstructed IP header.

**IP\_FORWARDING** is set by `ip_forward` and `ip_mforward` (multicast packet forwarding) and prevents `ip_output` from resetting any of the IP header fields.

The `MSG_DONTROUTE` flag to `send`, `sendto`, and `sendmsg` enables `IP_ROUTETOIF` for a single write ([Section 16.4](#)) while the `SO_DONTROUTE` socket option enables `IP_ROUTETOIF` for *all* writes on a particular socket ([Section 8.8](#)). The flag is passed by each of the transport protocols to `ip_output`.

The `IP_ALLOWBROADCAST` flag can be set by the `SO_BROADCAST` socket option ([Section 8.8](#)) but is passed only by UDP. The raw IP protocol sets `IP_ALLOWBROADCAST` by default. TCP does not support broadcasts, so `IP_ALLOWBROADCAST` is not passed by TCP to `ip_output`. There is no per-request flag for broadcasting.

## Construct IP header

60-73

If the caller provides any IP options they are merged with the packet by `ip_insertoptions` ([Section 9.8](#)), which returns the new header length.

We'll see in [Section 8.8](#) that a process can set the `IP_OPTIONS` socket option to specify the IP options for a socket. The transport layer for the socket (TCP or UDP) always passes these options to `ip_output`.

The IP header of a forwarded packet (`IP_FORWARDING`) or a packet with a preconstructed header (`IP_RAWOUTPUT`) should not be modified by `ip_output`. Any other packet (e.g., a UDP or TCP packet that originates at this host) needs to have several IP header fields initialized.

`ip_output` sets `ip_v` to 4 (`IPVERSION`), clears `ip_off` except for the DF bit, which is left as provided by the caller ([Chapter 10](#)), and assigns a unique identifier to `ip->ip_id` from the global integer `ip_id`, which is immediately incremented. Remember that

`ip_id` was seeded from the system clock during protocol initialization ([Section 7.8](#)). `ip_hl` is set to the header length measured in 32-bit words.

Most of the remaining fields in the IP headerlength, offset, TTL, protocol, TOS, and the destination address have already been initialized by the transport protocol. The source address may not be set, in which case it is selected after a route to the destination has been located ([Figure 8.25](#)).

## Packet already includes header

74-76

For a forwarded packet (or a raw IP packet with a header), the header length (in bytes) is saved in `hlen` for use by the fragmentation algorithm.

## Route Selection

After completing the IP header, the next task for `ip_output` is to locate a route to

the destination. This is shown in [Figure 8.24](#).

## Figure 8.24. ip\_output continued.

---

```
ip_output.c
77  /*
78   * Route packet.
79   */
80  if (ro == 0) {
81      ro = &iproute;
82      bzero((caddr_t) ro, sizeof(*ro));
83  }
84  dst = (struct sockaddr_in *) &ro->ro_dst;
85  /*
86   * If there is a cached route,
87   * check that it is to the same destination
88   * and is still up.  If not, free it and try again.
89   */
```

```

90     if (ro->ro_rt && ((ro->ro_rt->rt_flags & RTF_UP) == 0 ||
91         dst->sin_addr.s_addr != ip->ip_dst.s_addr)) {
92         RTFREE(ro->ro_rt);
93         ro->ro_rt = (struct rtentry *) 0;
94     }
95     if (ro->ro_rt == 0) {
96         dst->sin_family = AF_INET;
97         dst->sin_len = sizeof(*dst);
98         dst->sin_addr = ip->ip_dst;
99     }
100    /*
101     * If routing to interface only,
102     * short circuit routing lookup.
103     */
104 #define ifatoia(ifa)    ({struct in_ifaddr *)(ifa)})
105 #define sintosa(sin)   ({struct sockaddr *)(sin)})
106     if (flags & IP_ROUTETOIF) {
107         if ((ia = ifatoia(ifa_ifwithdstaddr(sintosa(dst)))) == 0 &&
108             (ia = ifatoia(ifa_ifwithnet(sintosa(dst)))) == 0) {
109             ipstat.ips_noroute++;
110             error = ENETUNREACH;
111             goto bad;
112         }
113         ifp = ia->ia_ifp;
114         ip->ip_ttl = 1;
115     } else {
116         if (ro->ro_rt == 0)
117             rtaalloc(ro);
118         if (ro->ro_rt == 0) {
119             ipstat.ips_noroute++;
120             error = EHOSTUNREACH;
121             goto bad;
122         }
123         ia = ifatoia(ro->ro_rt->rt_ifa);
124         ifp = ro->ro_rt->rt_ifp;
125         ro->ro_rt->rt_use++;
126         if (ro->ro_rt->rt_flags & RTF_GATEWAY)
127             dst = (struct sockaddr_in *) ro->ro_rt->rt_gateway;
128     }

```

/\* multicast destination (Figure 12.40) \*/

---

*ip\_output.c*

## Verify cached route

77-99

A cached route may be provided to `ip_output` as the `ro` argument. In [Chapter 24](#) we'll see that UDP and TCP maintain a route cache associated with each socket. If

a route has not been provided, ip\_output sets ro to point to the temporary route structure iproute.

If the cached destination is not to the current packet's destination, the route is discarded and the new destination address placed in dst.

## Bypass routing

100-114

A caller can prevent packet routing by setting the IP\_ROUTETOIF flag ([Section 8.8](#)). If this flag is set, ip\_output must locate an interface directly connected to the destination network specified in the packet. ifa\_ifwithdstaddr searches point-to-point interfaces, while in\_ifwithnet searches all the others. If neither function finds an interface connected to the destination network, ENETUNREACH is returned; otherwise, ifp points to the selected interface.

This option allows routing protocols to

bypass the local routing tables and force the packets to exit the system by a particular interface. In this way, routing information can be exchanged with other routers even when the local routing tables are incorrect.

## Locate route

115-122

If the packet is being routed (IP\_ROUTETOIF is off) and there is no cached route, rtalloc locates a route to the address specified by dst. ip\_output returns EHOSTUNREACH if rtalloc fails to find a route. If ip\_forward called ip\_output, EHOSTUNREACH is converted to an ICMP error. If a transport protocol called ip\_output, the error is passed back to the process ([Figure 8.21](#)).

123-128

ia is set to point to an address (the ifaddr structure) of the selected interface and ifp points to the interface's ifnet structure. If

the next hop is not the packet's final destination, dst is changed to point to the next-hop router instead of the packet's final destination. The destination address within the IP header remains unchanged, but the interface layer must deliver the packet to dst, the next-hop router.

## Source Address Selection and Fragmentation

The final section of ip\_output, shown in [Figure 8.25](#), ensures that the IP header has a valid source address and then passes the packet to the interface associated with the route. If the packet is larger than the interface's MTU, it must be fragmented and transmitted in pieces. As we did with the reassembly code, we omit the fragmentation code here and postpone discussion of it until [Chapter 10](#).

**Figure 8.25.** ip\_output continued.

---

```

212  /*
213   * If source address not specified yet, use address
214   * of outgoing interface.
215   */
216  if (ip->ip_src.s_addr == INADDR_ANY)
217      ip->ip_src = IA_SIN(ia)->sin_addr;
218  /*
219   * Look for broadcast address and
220   * verify user is allowed to send
221   * such a packet.
222   */

223  if (in_broadcast(dst->sin_addr, ifp)) {
224      if ((ifp->if_flags & IFF_BROADCAST) == 0) { /* interface check */
225          error = EADDRNOTAVAIL;
226          goto bad;
227      }
228      if ((flags & IP_ALLOWBROADCAST) == 0) { /* application check */
229          error = EACCES;
230          goto bad;
231      }
232      /* don't allow broadcast messages to be fragmented */
233      if ((u_short) ip->ip_len > ifp->if_mtu) {
234          error = EMSGSIZE;
235          goto bad;
236      }
237      m->m_flags |= M_BCAST;
238  } else
239      m->m_flags &= ~M_BCAST;

240  sendit:
241  /*
242   * If small enough for interface, can just send directly.
243   */
244  if ((u_short) ip->ip_len <= ifp->if_mtu) {
245      ip->ip_len = htons((u_short) ip->ip_len);
246      ip->ip_off = htons((u_short) ip->ip_off);
247      ip->ip_sum = 0;
248      ip->ip_sum = in_cksum(m, hlen);
249      error = (*ifp->if_output) (ifp, m,
250                                  (struct sockaddr *) dst, ro->ro_rt);
251      goto done;
252  }

    /* fragmentation (Section 10.3) */

339  done:
340  if (ro == &iproute && (flags & IP_ROUTETOIF) == 0 && ro->ro_rt)
341      RTFREE(ro->ro_rt);
342  return (error);
343  bad:
344  m_freem(m0);
345  goto done;
346 }

```

---

## Select source address

212-239

If ip\_src has not been specified, then ip\_output selects ia, the IP address of the outgoing interface, as the source address. This couldn't be done earlier when the other IP header fields were filled in because a route hadn't been selected yet. Forwarded packets always have a source address, but packets that originate at the local host may not if the sending process has not explicitly selected one.

If the destination IP address is a broadcast address, the interface must support broadcasting (IFF\_BROADCAST, [Figure 3.7](#)), the caller must explicitly enable broadcasting (IP\_ALLOWBROADCAST, [Figure 8.23](#)), and the packet must be small enough to be sent without fragmentation.

This last test is a policy decision. Nothing in the IP protocol specification explicitly prohibits the fragmentation of broadcast packets. By requiring the packet to fit within the MTU of the interface, however, there is an increased chance that the broadcast packet will be received at every

interface, because there is a better chance of receiving one undamaged packet than of receiving two or more undamaged packets.

If any of these conditions are not met, the packet is dropped and EADDRNOTAVAIL, EACCES, or EMSGSIZE is returned to the caller. Otherwise, M\_BCAST is set on the outgoing packet, which tells the interface output function to send the packet as a link-level broadcast. In [Section 21.10](#) we'll see that arpresolve translates the IP broadcast address to the Ethernet broadcast address.

If the destination address is not a broadcast address, ip\_output clears M\_BCAST.

If M\_BCAST were not cleared, the reply to a request packet that arrived as a broadcast might be accidentally returned as a broadcast. We'll see in [Chapter 11](#) that ICMP replies are constructed within the request packet in this way as are TCP RST packets ([Section 26.9](#)).

## Send packet

240-252

If the packet is small enough for the selected interface, `ip_len` and `ip_off` are converted to network byte order, the IP checksum is computed with `in_cksum` ([Section 8.7](#)), and the packet is passed to the `if_output` function of the selected interface.

## Fragment packet

253-338

Larger packets must be fragmented before they can be sent. We have omitted that code here and describe it in [Chapter 10](#) instead.

## Cleanup

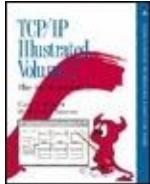
339-346

A reference count is maintained for the route entries. Recall that `ip_output` may

use a temporary route structure (iproute) if the argument ro is null. If necessary, RTFREE releases the route entry within iproute and decrements the reference count. The code at bad discards the current packet before returning.

Reference counting is a memory management technique. The programmer must count the number of external references to a data structure; when the count returns to 0, the memory can be safely returned to the free pool. Reference counting requires some discipline by the programmer, who must explicitly increase and decrease the reference count when appropriate.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.7 Internet Checksum: in\_cksum Function

Two operations dominate the time required to process packets: copying the data and computing checksums ([[Kay and Pasquale 1993](#)]). The flexible nature of the mbuf data structure is the primary method of reducing copy operations in Net/3. Efficient computing of checksums is harder since it is very hardware dependent. Net/3 contains several implementations of `in_cksum`.

**Figure 8.26. `in_cksum` versions in Net/3.**

Version	Source file
portable C	sys/netinet/in_cksum.c
SPARC	net3/sparc/sparc/in_cksum.c
68k	net3/luna68k/luna68k/in_cksum.c
VAX	sys/vax/vax/in_cksum.c
Tahoe	sys/tahoe/tahoe/in_cksum.c
HP 3000	sys/hp300/hp300/in_cksum.c
Intel 80386	sys/i386/i386/in_cksum.c

Even the portable C implementation has been optimized considerably. RFC 1071 [[Braden, Borman, and Partridge 1988](#)] and RFC 1141 [[Mallory and Kullberg 1990](#)] discuss the design and implementation of the Internet checksum function. RFC 1141 has been updated by RFC 1624 [[Rijsinghani 1994](#)]. From RFC 1071:

**1. Adjacent bytes to be checksummed are paired to form 16-bit integers, and the one's complement sum of these 16-bit integers is formed.**

- To generate a checksum, the checksum field itself is cleared, the 16-bit one's complement sum is computed over the bytes concerned, and the one's complement of this sum is placed in the checksum field.
- To verify a checksum, the one's

complement sum is computed over the same set of bytes, including the checksum field. If the result is all 1 bits (-0 in one's complement arithmetic, as explained below), the check succeeds.

Briefly, when addition is performed on integers in one's complement representation, the result is obtained by summing the two integers and adding any carry bit to the result to obtain the final sum. In one's complement arithmetic the negative of a number is formed by complementing each bit. There are two representations of 0 in one's complement arithmetic: all 0 bits, and all 1 bits. A more detailed discussion of one's complement representations and arithmetic can be found in [[Mano 1993](#)].

The checksum algorithm computes the value to place in the checksum field of the IP header before sending the packet. To compute this value, the checksum field in the header is set to 0 and the one's complement sum on the entire header (including options) is computed. The header is processed as an array of 16-bit

integers. Let's call the result of this computation  $a$ . Since the checksum field is explicitly set to 0,  $a$  is also the sum of all the IP header fields except the checksum. The one's complement of  $a$ , denoted  $-a$ , is placed in the checksum field and the packet is sent.

If no bits are altered in transit, the computed checksum at the destination should be the complement of  $(a+-a)$ . The sum  $(a+-a)$  in one's complement arithmetic is -0 (all 1 bits) and its complement is 0 (all 0 bits). So the computed checksum of an undamaged packet at the destination should always be 0. This is what we saw in [Figure 8.12](#). The following C code (which is not part of Net/3) is a naive implementation of this algorithm:

**Figure 8.27. A naive implementation of the IP checksum calculation.**

---

```
1 unsigned short
2 cksum(struct ip *ip, int len)
3 {
4     long      sum = 0;           /* assume 32 bit long, 16 bit short */
5     while (len > 1) {
6         sum += *((unsigned short *) ip)++;
7         if (sum & 0x80000000)    /* if high-order bit set, fold */
8             sum = (sum & 0xFFFF) + (sum >> 16);
9         len -= 2;
10    }
11    if (len)                  /* take care of left over byte */
12        sum += (unsigned short) *(unsigned char *) ip;
13    while (sum >> 16)
14        sum = (sum & 0xFFFF) + (sum >> 16);
15    return ~sum;
16 }
```

---

## 1-16

The only performance enhancement here is to accumulate the carry bits in the high-order 16 bits of sum. The accumulated carries are added to the low-order 16 bits when the loop terminates, until no more carries occur. RFC 1071 calls this *deferred carries*. This technique is useful on machines that don't have an add-with-carry instruction or when detecting a carry is expensive.

Now we show the portable C version from Net/3. It utilizes the deferred carry technique and works with packets stored in an mbuf chain.

42-140

Our naive checksum implementation assumed that all the bytes to be checksummed were in a contiguous buffer instead of in mbuf chains. This version of the checksum calculation handles the mbufs correctly using the same underlying algorithm: 16-bit words are summed in a 32-bit integer with the carries deferred. For mbufs with an odd number of bytes, the extra byte is saved and paired with the first byte of the next mbuf. Since unaligned access to 16-bit words is invalid or incurs a severe performance penalty on most architectures, a misaligned byte is saved and `in_cksum` continues adding with the next aligned word. `in_cksum` is careful to byte swap the sum when this occurs to ensure that even-numbered and odd-numbered data bytes are collected in separate sum bytes as required by the checksum algorithm.

## Loop unrolling

93-115

The three while loops in the function add

16 words, 4 words, and 1 word to the sum during each iteration. The unrolled loops reduce the loop overhead and can be considerably faster than a straightforward loop on some architectures. The price is increased code size and complexity.

**Figure 8.28. An optimized portable C implementation of the IP checksum calculation.**

```
42 #define ADDCARRY(x)  (x > 65535 ? x -= 65535 : x)
43 #define REDUCE {l_util.l = sum; sum = l_util.s[0] + l_util.s[1]; ADDCARRY(sum);}
44 int
45 in_cksum(m, len)
46 struct mbuf *m;
47 int      len;
48 {
49     u_short *w;
50     int      sum = 0;
51     int      mlen = 0;
52     int      byte_swapped = 0;
53     union {
54         char    c[2];
55         u_short s;
56     } s_util;
57     union {
58         u_short s[2];
59         long    l;
60     } l_util;
61     for (; m && len; m = m->m_next) {
62         if (m->m_len == 0)
63             continue;
64         w = mtod(m, u_short *);
65         if (mlen == -1) {
66             /*
67              * The first byte of this mbuf is the continuation of a
68              * word spanning between this mbuf and the last mbuf.
69              *
70              * s_util.c[0] is already saved when scanning previous mbuf.
71              */
72         s_util.c[1] = *(char *) w;
73         sum += s_util.s;
74         w = (u_short *) ((char *) w + 1);
75         mlen = m->m_len - 1;
76         len--;
77     } else
78         mlen = m->m_len;
79     if (len < mlen)
80         mlen = len;
81     len -= mlen;
82     /*
83      * Force to even boundary.
84      */
85     if ((l & (int) w) && (mlen > 0)) {
86         REDUCE;
87         sum <= 8;
88         s_util.c[0] = *(u_char *) w;
89         w = (u_short *) ((char *) w + 1);
90         mlen--;
91         byte_swapped = 1;
92     }
}
```

```

93      /*
94       * Unroll the loop to make overhead from
95       * branches &c small.
96       */
97     while ((mlen -= 32) >= 0) {
98         sum += w[0]; sum += w[1]; sum += w[2]; sum += w[3];
99         sum += w[4]; sum += w[5]; sum += w[6]; sum += w[7];
100        sum += w[8]; sum += w[9]; sum += w[10]; sum += w[11];
101        sum += w[12]; sum += w[13]; sum += w[14]; sum += w[15];
102        w += 16;
103    }
104    mlen += 32;
105    while ((mlen -= 8) >= 0) {
106        sum += w[0]; sum += w[1]; sum += w[2]; sum += w[3];
107        w += 4;
108    }
109    mlen += 8;
110    if (mlen == 0 && byte_swapped == 0)
111        continue;
112    REDUCE;
113    while ((mlen -= 2) >= 0) {
114        sum += *w++;
115    }
116    if (byte_swapped) {
117        REDUCE;
118        sum <= 8;
119        byte_swapped = 0;
120        if (mlen == -1) {
121            s_util.c[1] = *(char *) w;
122            sum += s_util.s;
123            mlen = 0;
124        } else
125            mlen = -1;
126        } else if (mlen == -1)
127            s_util.c[0] = *(char *) w;
128    }
129    if (len)
130        printf("cksum: out of data\n");
131    if (mlen == -1) {
132        /* The last mbuf has odd # of bytes. Follow the standard (the odd
133           byte may be shifted left by 8 bits or not as determined by
134           endian-ness of the machine) */
135        s_util.c[1] = 0;
136        sum += s_util.s;
137    }
138    REDUCE;
139    return (~sum & 0xffff);
140 }

```

*in\_cksum.c*

## More Optimizations

RFC 1071 mentions two optimizations that don't appear in Net/3: a combined copy-with-checksum operation and incremental checksum updates. Merging the copy and

checksum operations is not as important for the IP header checksum as it is for the TCP and UDP checksums, which cover many more bytes. This merged operation is discussed in [Section 23.12](#). [Partridge and Pink 1993] report that an inline version of the IP header checksum is faster than calling the more general `in_cksum` function and can be done in six to eight assembler instructions (for the standard 20-byte IP header).

The design of the checksum algorithm allows a packet to be changed and the checksum updated without reexamining all the bytes. RFC 1071 contains a brief discussion of this topic. RFCs 1141 and 1624 contain more detailed discussions. A typical use of this technique occurs during packet forwarding. In the common case, when a packet has no options, only the TTL field changes during forwarding. The checksum in this case can be recomputed by a single addition with an end-around carry.

In addition to being more efficient, an incremental checksum can help detect

headers corrupted by buggy software. A corrupted header is detected by the next system if the checksum is computed incrementally, but if it is recomputed from scratch, the checksum incorporates the erroneous bytes and the corrupted header is not detected by the next system. The end-to-end checksum used by UDP or TCP detects the error at the final destination. We'll see in [Chapters 23](#) and [25](#) that the UDP and TCP checksums incorporate several parts of the IP header.

For an example of the checksum function that utilizes hardware add-with-carry instructions to compute the checksum 32 bits at a time, see the VAX implementation of `in_cksum` in the file `sys/vax/in_cksum.c`.



## Chapter 8. IP: Internet Protocol

---

### 8.8 setsockopt and getsockopt System Calls

Net/3 provides access to several networking features via system calls. These system calls support a generic interface that aren't supported by the standard system calls.

```
int setsockopt (int s, int level, int option, ...);
```

```
int getsockopt (int s, int level, int option, ...);
```

Most socket options affect only the socket on which they are set. Some options affect the entire system. The socket options are described in [Chapter 12](#).

setsockopt and getsockopt set and get options according to the protocol associated with the socket. The possible values for *level* within the protocols that support them are:

**Figure 8.29. setsockopt options**

Domain	Protocol	level	Function	Reference
any	any	<i>SOL_SOCKET</i>	<i>so_setopt</i> and <i>so_getopt</i>	Figures 17.5 and 17.11
IP	UDP	<i>IPPROTO_IP</i>	<i>ip_ctloutput</i>	Figure 8.31
	TCP	<i>IPPROTO_TCP</i> <i>IPPROTO_IP</i>	<i>tcp_ctloutput</i> <i>ip_ctloutput</i>	Section 30.6 Figure 8.31
	raw IP ICMP IGMP	<i>IPPROTO_IP</i>	<i>rip_ctloutput</i> and <i>ip_ctloutput</i>	Section 32.8

We describe the implementation of the *setsockopt* options at the *IPPROTO\_IP* level, the implementation of individual options within the *IPPROTO\_IP* level, and the options that provide access to IP features.

Throughout the text we summarize socket options at the *IPPROTO\_IP* level. The option appears in the first column, the *optval* appears in the second column, and the function appears in the third column.

**Figure 8.30. Socket options: IPPROTO\_IP**

<i>optname</i>	<i>optval</i> type	Function	Description
<i>IP_OPTIONS</i>	<i>void *</i>	<i>in_pcbopts</i>	set or get IP options to be included in outgoing datagrams
<i>IP_TOS</i>	int	<i>ip_ctloutput</i>	set or get IP TOS for outgoing datagrams
<i>IP_TTL</i>	int	<i>ip_ctloutput</i>	set or get IP TTL for outgoing datagrams
<i>IP_RECVDSTADDR</i>	int	<i>ip_ctloutput</i>	enable or disable queueing of IP destination address (UDP only)
<i>IP_RECVOPTS</i>	int	<i>ip_ctloutput</i>	enable or disable queueing of incoming IP options as control information (UDP only, not implemented)
<i>IP_RECVRETOPTS</i>	int	<i>ip_ctloutput</i>	enable or disable queueing of reversed source route associated with incoming datagram (UDP only, not implemented)

[Figure 8.31](#) shows the overall organization of the IPPROTO\_IP options. In [Section 32.8](#) we show how to use them with sockets.

**Figure 8.31. ip\_output.c**

```
ip_output.c
431 int
432 ip_ctloutput(op, so, level, optname, mp)
433 int     op;
434 struct socket *so;
435 int     level, optname;
436 struct mbuf **mp;
437 {
438     struct inpcb *inp = sotoinpcb(so);
439     struct mbuf *m = *mp;
440     int     optval;
441     int     error = 0;
442     if (level != IPPROTO_IP) {
443         error = EINVAL;
444         if (op == PRCO_SETOPT && *mp)
445             (void) m_free(*mp);
446     } else
447         switch (op) {
448             case PRCO_SETOPT:
449                 switch (optname) {

/* PRCO_SETOPT processing (Figures 8.32 and 12.17) */

500                     freeit:
501                     default:
502                         error = EINVAL;
503                         break;
504                     }
505                     if (m)
506                         (void) m_free(m);
507                     break;
508                 case PRCO_GETOPT:
509                     switch (optname) {

/* PRCO_GETOPT processing (Figures 8.33 and 12.17) */

546                     default:
547                         error = ENOPROTOOPT;
548                         break;
549                     }
550                     break;
551                 }
552             return (error);
553 }
```

ip\_ctloutput's first argument, op, is either PRCC to the socket on which the request was issued or to retrieve, and mp points indirectly to an m initialized to point to the mbuf referenced by \*r

448-500

If an unrecognized option is specified in the call to ip\_ctloutput (e.g., to the switch), ip\_ctloutput releases any mbuf passed to it.

501-553

Unrecognized options passed to getsockopt are released by the caller; the mbuf is released by the caller.

## PRCO\_SETOPT Processing

The processing for PRCO\_SETOPT is shown in Figure 8.32.

**Figure 8.32. ip\_ctloutput**

```

450         case IP_OPTIONS:
451             return (ip_pcbopts(&inp->inp_options, m));
452         case IP_TOS:
453         case IP_TTL:
454         case IP_RECVOPTS:
455         case IP_RECVRETOPTS:
456         case IP_RECVDSTADDR:
457             if (m->m_len != sizeof(int))
458                 error = EINVAL;
459             else {
460                 optval = *mtod(m, int *);
461                 switch (optname) {
462                     case IP_TOS:
463                         inp->inp_ip.ip_tos = optval;
464                         break;
465                     case IP_TTL:
466                         inp->inp_ip.ip_ttl = optval;
467                         break;
468 #define OPTSET(bit) \
469     if (optval) \
470         inp->inp_flags |= bit; \
471     else \
472         inp->inp_flags &= ~bit;
473                     case IP_RECVOPTS:
474                         OPTSET(INP_RECVOPTS);
475                         break;
476                     case IP_RECVRETOPTS:
477                         OPTSET(INP_RECVRETOPTS);
478                         break;
479                     case IP_RECVDSTADDR:
480                         OPTSET(INP_RECVDSTADDR);
481                         break;
482                 }
483             }
484         break;

```

---

ip\_output.c

## 450-451

IP\_OPTIONS is processed by ip\_pcbopts ([Figure 450-451](#))

## 452-484

The IP\_TOS, IP\_TTL, IP\_RECVOPTS, IP\_RECVRETOPTS, and IP\_RECVDSTADDR options are processed by ip\_pcbopts. These options are to be available in the mbuf pointed to by m. The ip\_tos or ip\_ttl values associated with the socket are set to optval if optval is nonzero (or 0).

Figure 8.30 showed that IP\_RECVOPTS and IF see that the settings of these options are ignc

## PRCO\_GETOPT Processing

Figure 8.33 shows the code that retrieves the I

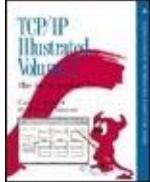
Figure 8.33. ip\_ctloutput

```
ip_output.c
503     case IP_OPTIONS:
504         *mp = m = m_get(M_WAIT, MT_SOOPTS);
505         if (inp->inp_options) {
506             m->m_len = inp->inp_options->m_len;
507             bcopy(mtod(inp->inp_options, caddr_t),
508                   mtod(m, caddr_t), (unsigned) m->m_len);
509         } else
510             m->m_len = 0;
511         break;
512
513     case IP_TOS:
514     case IP_TTL:
515     case IP_RECVOPTS:
516     case IP_RECVRETOPTS:
517     case IP_RECVDSTADDR:
518         *mp = m = m_get(M_WAIT, MT_SOOPTS);
519         m->m_len = sizeof(int);
520         switch (optname) {
521
522             case IP_TOS:
523                 optval = inp->inp_ip.ip_tos;
524                 break;
525
526 #define OPTBIT(bit) (inp->inp_flags & bit ? 1 : 0)
527
528             case IP_RECVOPTS:
529                 optval = OPTBIT(INP_RECVOPTS);
530                 break;
531
532             case IP_RECVRETOPTS:
533                 optval = OPTBIT(INP_RECVRETOPTS);
534                 break;
535
536             case IP_RECVDSTADDR:
537                 optval = OPTBIT(INP_RECVDSTADDR);
538                 break;
539         }
540         *mtod(m, int *) = optval;
541         break;
```

## 503-538

For IP\_OPTIONS, ip\_ctloutput returns an mbuf  
For the remaining options, ip\_ctloutput returns  
with the option. The value is returned in the ml  
is on (or off) in inp\_flags.

Notice that the IP options are stored in the prot  
socket.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.9 ip\_sysctl Function

Figure 7.27 showed that the ip\_sysctl function is called when the protocol and family identifiers are 0 in a call to sysctl. Figure 8.34 shows the three parameters supported by ip\_sysctl.

**Figure 8.34. ip\_sysctl parameters.**

sysctl constant	Net/3 variable	Description
<code>IPCTL_FORWARDING</code>	<code>ipforwarding</code>	Should the system forward IP packets?
<code>IPCTL_SENDREDIRECTS</code>	<code>ipsendredirects</code>	Should the system send ICMP redirects?
<code>IPCTL_DEFTTL</code>	<code>ip_defttl</code>	Default TTL for IP packets.

Figure 8.35 shows the ip\_sysctl function.

**Figure 8.35. ip\_sysctl function.**

```
984 int
985 ip_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
986 int     *name;
987 u_int    namelen;
988 void    *oldp;
989 size_t   *oldlenp;
990 void    *newp;
991 size_t   newlen;
992 {
993     /* All sysctl names at this level are terminal. */
994     if (namelen != 1)
995         return (ENOTDIR);
996
997     switch (name[0]) {
998         case IPCTL_FORWARDING:
999             return (sysctl_int(oldp, oldlenp, newp, newlen, &ipforwarding));
1000        case IPCTL_SENDREDIRECTS:
1001            return (sysctl_int(oldp, oldlenp, newp, newlen,
1002                             &ipsendredirects));
1003        case IPCTL_DEFTTL:
1004            return (sysctl_int(oldp, oldlenp, newp, newlen, &ip_defttl));
1005        default:
1006            return (EOPNOTSUPP);
1007    }
1008 }  
ip_input.c
```

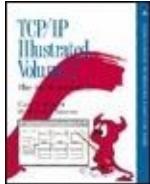
## 984-995

Since `ip_sysctl` does not forward `sysctl` requests to any other functions, there can be only one remaining component in `name`. If not, `ENOTDIR` is returned.

## 996-1008

The switch statement selects the appropriate call to `sysctl_int`, which accesses or modifies `ipforwarding`, `ipsendredirects`, or `ip_defttl`. `EOPNOTSUPP` is returned for unrecognized options.

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 8. IP: Internet Protocol

### 8.10 Summary

IP is a best-effort datagram service that provides the delivery mechanism for all other Internet protocols. The standard IP header is 20 bytes long, but may be followed by up to 40 bytes of options. IP can split large datagrams into fragments to be transmitted and reassembles the fragments at the final destination. Option processing is discussed in [Chapter 9](#), and fragmentation and reassembly is discussed in [Chapter 10](#).

ipintr ensures that IP headers have arrived undamaged and determines if they have arrived at their final destination by comparing the destination address to the

IP addresses of the system's interfaces and to several broadcast addresses. ipintr passes datagrams that have reached their final destination to the transport protocol specified within the packet. If the system is configured as a router, datagrams that have not reached their final destination are sent to ip\_forward for routing toward their final destination. Packets have a limited lifetime. If the TTL field drops to 0, the packet is dropped by ip\_forward.

The Internet checksum function is used by many of the Internet protocols and implemented by in\_cksum in Net/3. The IP checksum covers only the header (and options), not the data, which must be protected by checksums at the transport protocol level. As one of the most time-consuming operations in IP, the checksum function is often optimized for each platform.

## Exercises

Should IP accept broadcast packets

**8.1** when there are no IP addresses assigned to any interfaces?

**8.2** Modify ip\_forward and ip\_output to do an incremental update of the IP checksum when a packet without options is being forwarded.

**8.3** Why is it necessary to check for a link-level broadcast (M\_BCAST flag in an mbuf) and for an IP-level broadcast (in\_canforward) when rejecting packets for forwarding? When would a packet arrive as a link-level broadcast but with an IP unicast destination?

**8.4** Why isn't an error message returned to the sender when an IP packet arrives with checksum errors?

Assume that a process on a multihomed host has selected an explicit source address for its

outgoing packets. Furthermore, assume that the packet's destination **8.5** is reached through an interface other than the one selected as the packet's source address. What happens when the first-hop router discovers that the packets should be going through a different router? Is a redirect message sent to the host?

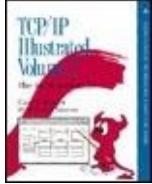
A new host is attached to a subnetted network and is configured to perform routing (ipforwarding **8.6** equals 1) but its network interface has not been assigned a subnet mask. What happens when this host receives a subnet broadcast packet?

Why is it necessary to decrement **8.7** ip\_ttl after testing it (versus before) in [Figure 8.17](#)?

What would happen if two routers **8.8** each considered the other the best next-hop destination for a packet?

Which addresses would not be checked in [Figure 8.14](#) for a packet **8.9** arriving at the SLIP interface? Would any additional addresses be checked that aren't listed in [Figure 8.14](#)?

`ip_forward` converts the fragment id from host byte order to network byte **8.10** order before calling `icmp_error`. Why does it not also convert the fragment offset?



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 9. IP Option Processing

[Section 9.1. Introduction](#)

[Section 9.2. Code Introduction](#)

[Section 9.3. Option Format](#)

[Section 9.4. ip\\_dooptions Function](#)

[Section 9.5. Record Route Option](#)

[Section 9.6. Source and Record Route Options](#)

[Section 9.7. Timestamp Option](#)

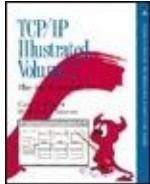
[Section 9.8. ip\\_insertoptions Function](#)

[Section 9.9. ip\\_pcbopts Function](#)

[Section 9.10. Limitations](#)

[Section 9.11. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

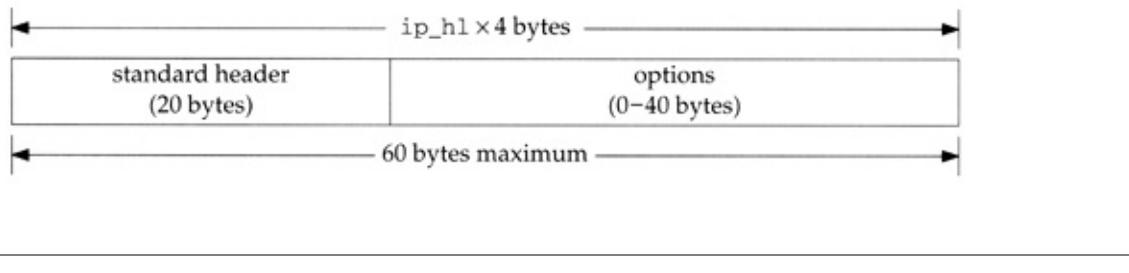
### 9.1 Introduction

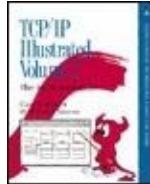
Recall from [Chapter 8](#) that the IP input function (ipintr) processes options after it verifies the packet's format (checksum, length, etc.) and before it determines whether the packet has reached its final destination. This implies that a packet's options are processed by every router it encounters and by the final destination host.

RFCs 791 and 1122 specify the IP options and processing rules. This chapter describes the format and processing of most IP options. We'll also show how a transport protocol can specify the IP options to be included in an IP datagram.

An IP packet can include optional fields that are processed before the packet is forwarded or accepted by a system. An IP implementation can handle options in any order; for Net/3, it is the order in which the options appear in the packet. [Figure 9.1](#) shows that up to 40 bytes of options may follow the standard IP header.

## Figure 9.1. An IP header may contain 0 to 40 bytes of IP options.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.2 Code Introduction

Two headers describe the data structures for IP options. Option processing code is found in two C files. [Figure 9.2](#) lists the relevant files.

**Figure 9.2. Files discussed in this chapter.**

File	Description
netinet/ip.h	ip_timestamp structure
netinet/ip_var.h	ipoption structure
netinet/ip_input.c	option processing
netinet/ip_output.c	ip_insertoptions function

### Global Variables

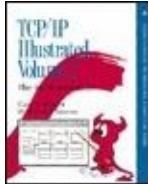
The two global variables described in [Figure 9.3](#) support the reversal of source routes.

## **Figure 9.3. Global variables introduced in this chapter.**

Variable	Datatype	Description
ip_nhops	int	hop count for previous source route
ip_srcrt	struct ip_srcrt	previous source route

## **Statistics**

The only statistic updated by the options processing code is ips\_badoptions from the ipstat structure, which [Figure 8.4](#) described.



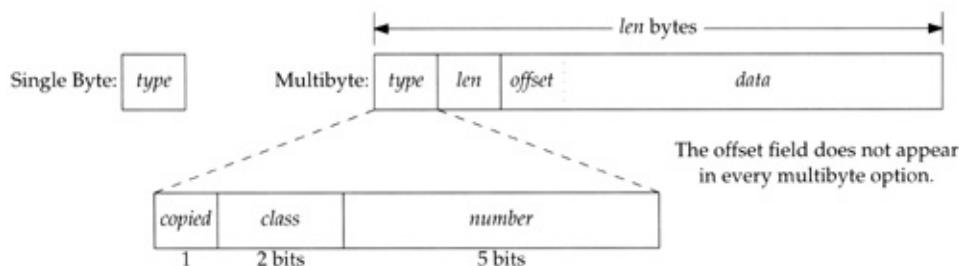
TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.3 Option Format

The IP option field may contain 0 or more individual options. The two types of options, single-byte and multibyte, are illustrated in [Figure 9.4](#).

**Figure 9.4. The organization of single-byte and multibyte IP options.**



All options start with a 1-byte *type* field. In

multibyte options, the *type* field is followed immediately by a *len* field, and the remaining bytes are the *data*. The first byte of the *data* field for many options is a 1-byte *offset* field, which points to a byte within the *data* field. The *len* byte covers the *type*, *len*, and *data* fields in its count. The *type* is further divided into three internal fields: a 1-bit *copied* flag, a 2-bit *class* field, and a 5-bit *number* field. [Figure 9.5](#) lists the currently defined IP options. The first two options are single-byte options; the remainder are multibyte options.

## Figure 9.5. IP options defined by RFC 791.

Constant	Type		Length (bytes)	Net/3	Description
	Decimal	Binary			
<i>IPOPT_EOL</i>	0-0-0	0	0-00-00000	1	• end of option list (EOL)
<i>IPOPT_NOP</i>	0-0-1	1	0-00-00001	1	• no operation (NOP)
<i>IPOPT_RR</i>	0-0-7	7	0-00-00111	varies	• record route
<i>IPOPT_TS</i>	0-2-4	68	0-10-00100	varies	• timestamp
<i>IPOPT_SECURITY</i>	1-0-2	130	1-00-00010	11	• basic security
<i>IPOPT_LSRR</i>	1-0-3	131	1-00-00011	varies	• loose source and record route (LSRR)
<i>IPOPT_SATID</i>	1-0-5	133	1-00-00101	varies	extended security
<i>IPOPT_SSRR</i>	1-0-8	136	1-00-01000	4	stream identifier
	1-0-9	137	1-00-01001	varies	• strict source and record route (SSRR)

The first column shows the Net/3 constant for the option, followed by the decimal and

binary values of the type in columns 2 and 3, and the expected length of the option in column 4. The Net/3 column shows those options that are implemented in Net/3 by ip\_dooptions. IP must silently ignore any option it does not understand. We don't describe the options that are not implemented in Net/3: security and stream ID. The stream ID option is obsolete and the security options are used primarily by the U.S. military. See RFC 791 for more information.

Net/3 examines the *copied* flag when it fragments a packet with options ([Section 10.4](#)). The flag indicates whether the individual option should be copied into the IP header of the fragments. The *class* field groups related options as described in [Figure 9.6](#). All the options in [Figure 9.5](#) have a *class* of 0 except for the timestamp option, which has a *class* of 2.

**Figure 9.6. The class field within an IP option.**

<i>class</i>	Description
0	control
1	reserved
2	debugging and measurement
3	reserved

---

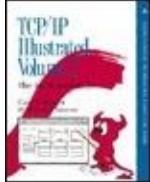
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.4 ip\_dooptions Function

In [Figure 8.13](#) we saw that ipintr calls ip\_dooptions just before it checks the destination address of the packet. ip\_dooptions is passed a pointer, m, to a packet and processes the options it knows about. If ip\_dooptions forwards the packet, as can happen with the LSRR and SSRR options, or discards the packet because of an error, it returns 1. If it doesn't forward the packet, ip\_dooptions returns 0 and ipintr continues processing the packet.

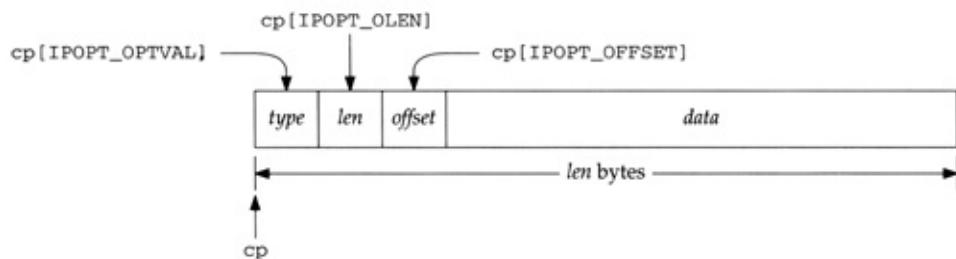
ip\_dooptions is a long function, so we show it in parts. The first part initializes a for loop to process each option in the

header.

When processing an individual option, *cp* points to the first byte of the option.

[Figure 9.7](#) illustrates how the *type*, *length*, and, when applicable, the *offset* fields are accessed with constant offsets from *cp*.

### Figure 9.7. Access to IP option fields is by constant offsets.



The RFCs refer to the *offset* field as a *pointer*, which is slightly more descriptive than the term *offset*. The value of *offset* is the index (starting with *type* at index 1) of a byte within the option, and not a 0-based offset from *type*. The minimum value for *offset* is 4 (IPOPT\_MINOFF), which points to the first byte of the *data* field in a multibyte option.

[Figure 9.8](#) shows the overall organization of the ip\_dooptions function.

## Figure 9.8. ip\_dooptions function.

```
ip_input.c
553 int
554 ip_dooptions(m)
555 struct mbuf *m;
556 {
557     struct ip *ip = mtod(m, struct ip *);
558     u_char *cp;
559     struct ip_timestamp *ipt;
560     struct in_ifaddr *ia;
561     int opt, optlen, cnt, off, code, type = ICMP_PARAMPROB, forward = 0;
562     struct in_addr *sin, dst;
563     n_time ntime;
564
565     dst = ip->ip_dst;
566     cp = (u_char *) (ip + 1);
567     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
568     for (; cnt > 0; cnt -= optlen, cp += optlen) {
569         opt = cp[IPOPT_OPTVAL];
570         if (opt == IPOPT_EOL)
571             break;
572         if (opt == IPOPT_NOP)
573             optlen = 1;
574         else {
575             optlen = cp[IPOPT_OLEN];
576             if (optlen <= 0 || optlen > cnt) {
577                 code = &cp[IPOPT_OLEN] - (u_char *) ip;
578                 goto bad;
579             }
580             switch (opt) {
581             default:
582                 break;
583
584             /* option processing */
585
586         }
587     }
588     if (forward) {
589         ip_forward(m, 1);
590         return (1);
591     }
592     return (0);
593
594     bad:
595     ip->ip_len -= ip->ip_hl << 2; /* XXX icmp_error adds in hdr length */
596     icmp_error(m, type, code, 0, 0);
597     ipstat.ips_badoptions++;
598     return (1);
599 }
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730 )
```

ip\_dooptions initializes the ICMP error type, type, to ICMP\_PARAMPROB, which is a generic value for any error that does not have a specific error type of its own. For ICMP\_PARAMPROB, code is the offset within the packet of the erroneous byte. This is the default ICMP error message; some options change these values.

ip points to an ip structure with a size of 20 bytes, so ip+1 points to the next ip structure following the IP header. Since ip\_dooptions wants the address of the *byte* after the IP header, the cast converts the resulting pointer to a pointer to an unsigned byte (u\_char). Therefore cp points to the first byte beyond the standard IP header, which is the first byte of the IP options.

## EOL and NOP processing

567-582

The for loop processes each option in the order it appears in the packet. An EOL option terminates the loop, as does an

invalid option length (i.e., the option length indicates that the option data extends beyond the IP header). A NOP option is skipped when it appears. The default case for the switch statement implements the requirement that a system ignore unknown options.

The following sections describe each of the options handled within the switch statement. If ip\_dooptions processes all the options in the packet without finding an error, control falls through to the code after the switch.

## Source route forwarding

719-724

If the packet needs to be forwarded, forward is set by the SSRR or LSRR option processing code. The packet is passed to ip\_forward with a 1 as the second argument to specify that the packet is source routed.

Recall from [Section 8.5](#) that ICMP

redirects are not generated for source-routed packets; this is the reason for the second argument to ip\_forward.

ip\_dooptions returns 1 if the packet has been forwarded. If the packet does not include a source route, 0 is returned to ipintr to indicate that the datagram needs further processing. Note that source route forwarding occurs whether the system is configured as a router (ipforwarding equals 1) or not.

This is a somewhat controversial policy, but is mandated by RFC 1122. RFC 1127 [[Braden 1989c](#)] describes this as an open issue.

## Error handling

725-730

If an error occurs within the switch, ip\_dooptions jumps to bad. The IP header length is subtracted from the packet length since icmp\_error assumes the header length is not included in the packet length.

icmp\_error sends the appropriate error message, and ip\_dooptions returns 1 to prevent ipintr from processing the discarded packet.

The following sections describe each of the options that are processed by Net/3.

---

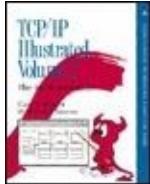
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

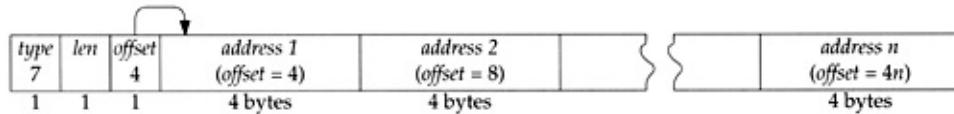
## Chapter 9. IP Option Processing

### 9.5 Record Route Option

The record route option causes the route taken by a packet to be recorded within the packet as it traverses an internet. The size of the option is fixed by the source host when it constructs the option and must be large enough to hold all the expected addresses. Recall that only 40 bytes of options may appear in an IP packet. The record route option has 3 bytes of overhead followed by a list of addresses (4 bytes each). If it is the only option, up to 9 ( $3 + 4 \times 9 = 39$ ) addresses may appear. Once the allocated space in the option has been filled, the packet is forwarded as usual but no more addresses are recorded by the intermediate systems.

Figure 9.9 illustrates the format of a record route option and Figure 9.10 shows the source code.

## Figure 9.9. The record route option. $n$ must be $\leq 9$ .



## Figure 9.10. ip\_dooptions function: record route option processing.

```
647     case IPOPT_RR:
648         if ((off = cp[IPOPT_OFFSET]) < IPOPT_MINOFF) {
649             code = &cp[IPOPT_OFFSET] - (u_char *) ip;
650             goto bad;
651         }
652         /*
653          * If no space remains, ignore.
654          */
655         off--;           /* 0 origin */
656         if (off > optlen - sizeof(struct in_addr))
657             break;
658         bcopy((caddr_t) (&ip->ip_dst), (caddr_t) & ipaddr.sin_addr,
659               sizeof(ipaddr.sin_addr));
660         /*
661          * locate outgoing interface; if we're the destination,
662          * use the incoming interface (should be same).
663          */
664         if ((ia = (INA) ifa_ifwithaddr((SA) & ipaddr)) == 0 &&
665             (ia = ip_rtaddr(ipaddr.sin_addr)) == 0) {
666             type = ICMP_UNREACH;
667             code = ICMP_UNREACH_HOST;
668             goto bad;
669         }
670         bcopy((caddr_t) & (IA_SIN(ia)->sin_addr),
671               (caddr_t) (cp + off), sizeof(struct in_addr));
672         cp[IPOPT_OFFSET] += sizeof(struct in_addr);
673         break;
```

647-657

If the option offset is too small, ip\_dooptions sends an ICMP parameter problem error. The variable code is set to the byte offset of the invalid option offset within the packet, and the ICMP parameter problem error has this code value when the error is generated at the label bad ([Figure 9.8](#)). If there is no space in the option for additional addresses, the option is ignored and processing continues with the next option.

## Record address

658-673

If ip\_dst is one of the systems addresses (the packet has arrived at its destination), the address of the receiving interface is recorded in the option; otherwise the address of the outgoing interface as provided by ip\_rtaddr is recorded. (The INA and SA macros are defined in [Figure 9.15](#).) The offset is updated to point to the next available address position in the

option. If ip\_rtaddr can't find a route to the destination, an ICMP host unreachable error is sent.

Section 7.3 of Volume 1 contains examples of the record route option.

## ip\_rtaddr Function

The ip\_rtaddr function consults a route cache and, if necessary, the complete routing tables to locate a route to a given IP address. It returns a pointer to the in\_ifaddr structure associated with the outgoing interface for the route. The function is shown in [Figure 9.11](#).

**Figure 9.11. ip\_rtaddr function: locate outgoing interface.**

```
735 struct in_ifaddr *
736 ip_rtaddr(dst)
737 struct in_addr dst;
738 {
739     struct sockaddr_in *sin;
740     sin = (struct sockaddr_in *) &ipforward_rt.ro_dst;
741     if (ipforward_rt.ro_rt == 0 || dst.s_addr != sin->sin_addr.s_addr) {
742         if (ipforward_rt.ro_rt) {
743             RTFREE(ipforward_rt.ro_rt);
744             ipforward_rt.ro_rt = 0;
745         }
746         sin->sin_family = AF_INET;
747         sin->sin_len = sizeof(*sin);
748         sin->sin_addr = dst;
749         rtalloc(&ipforward_rt);
750     }
751     if (ipforward_rt.ro_rt == 0)
752         return ((struct in_ifaddr *) 0);
753     return ((struct in_ifaddr *) ipforward_rt.ro_rt->rt_ifa);
754 }
```

ip\_input.c

## Check IP forwarding cache

735-741

If the route cache is empty, or if dest, the only argument to ip\_rtaddr, does not match the destination in the route cache, the routing tables must be consulted to select an outgoing interface.

## Locate route

742-750

The old route (if any) is discarded and the new destination address is stored in \*sin (which is the ro\_dst member of the

forwarding cache), rtalloc searches the routing tables for a route to the destination.

## Return route information

751-754

If no route is available, a null pointer is returned. Otherwise, a pointer to the interface address structure associated with the selected route is returned.

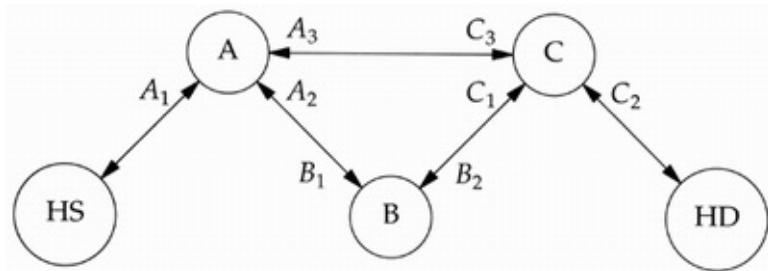
## Chapter 9. IP Option Processing

### 9.6 Source and Record Route Options

Normally a packet is forwarded along a path chosen by the router. Source and record route options allow the source host to specify a route that overrides routing decisions of the intermediate routers as the packet travels toward its destination.

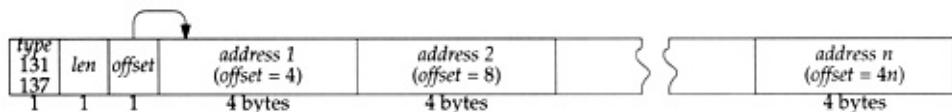
A *strict* route includes the address of every intermediate system and destination; a *loose* route specifies only some of the intermediate systems. A packet can choose any path between two systems listed in a strict route. Only those paths that pass through all the systems listed in a loose route are allowed between the systems listed in a strict route. Figure 9.12 illustrates source route processing.

**Figure 9.12. Source Route Processing**



A, B, and C are routers and HS and HD are the interface has its own IP address, we see that routers A, B, and C have multiple addresses and record route options.

**Figure 9.13.** The loose and strict modes of the `Object` class.



The source and destination addresses in the IP option specify the route and the packet's current location. How this information changes as the packet follows the path from C to HD. The loose source route specified by the IP header is shown in the first column. Each row represents the state of the packet as it passes through a router or host. The last line shows the packet as it appears at the destination host.

**Figure 9.14.** The source route option is

System	IP Header		Source Route Option		
	ip_src	ip_dst	offset	addresses	
HS	HS	$A_3$	4	$\bullet$	$B_1 \quad C_1 \quad HD$
A	HS	$B_1$	8	$A_2$	$\bullet \quad C_1 \quad HD$
B	HS	$C_1$	12	$A_2$	$B_2 \quad \bullet \quad HD$
C	HS	HD	16	$A_2$	$B_2 \quad C_2 \quad \bullet$
HD	HS	HD	16	$A_2$	$B_2 \quad C_2 \quad \bullet$

**Figure 9.15. ip\_dooptions function:**

```

583         /*
584          * Source routing with record.
585          * Find interface with current destination address.
586          * If none on this machine then drop if strictly routed,
587          * or do nothing if loosely routed.
588          * Record interface address and bring up next address
589          * component. If strictly routed make sure next
590          * address is on directly accessible net.
591          */
592     case IPOPT_LSRR:
593     case IPOPT_SSRR:
594         if ((off = cp[IPOPT_OFFSET]) < IPOPT_MINOFF) {
595             code = &cp[IPOPT_OFFSET] - (u_char *) ip;
596             goto bad;
597         }
598         ipaddr.sin_addr = ip->ip_dst;
599         ia = (struct in_ifaddr *)
600             ifa_ifwithaddr((struct sockaddr *) &ipaddr);
601         if (ia == 0) {
602             if (opt == IPOPT_SSRR) {
603                 type = ICMP_UNREACH;
604                 code = ICMP_UNREACH_SRCFAIL;
605                 goto bad;
606             }
607             /*
608              * Loose routing, and not at next destination
609              * yet; nothing to do except forward.
610              */
611             break;
612         }
613         off--;           /* 0 origin */
614         if (off > optlen - sizeof(struct in_addr)) {
615             /*
616              * End of source route. Should be for us.
617              */
618             save_rte(cp, ip->ip_src);
619             break;
620         }
621         /*
622          * locate outgoing interface
623          */
624         bcopy((caddr_t) (cp + off), (caddr_t) & ipaddr.sin_addr,
625               sizeof(ipaddr.sin_addr));
626         if (opt == IPOPT_SSRR) {
627 #define INA struct in_ifaddr *
628 #define SA struct sockaddr *
629             if ((ia = (INA) ifa_ifwithdstaddr((SA) & ipaddr)) == 0)
630                 ia = (INA) ifa_ifwithnet((SA) & ipaddr);
631             } else
632                 ia = ip_rtaddr(ipaddr.sin_addr);
633             if (ia == 0) {
634                 type = ICMP_UNREACH;
635                 code = ICMP_UNREACH_SRCFAIL;

636                 goto bad;
637             }
638             ip->ip_dst = ipaddr.sin_addr;
639             bcopy((caddr_t) & (IA_SIN(ia)->sin_addr),
640                   (caddr_t) (cp + off), sizeof(struct in_addr));
641             cp[IPOPT_OFFSET] += sizeof(struct in_addr);
642             /*
643              * Let ip_intr's mcast routing check handle mcast pkts
644              */
645             forward = !IN_MULTICAST(ntohl(ip->ip_dst.s_addr));
646             break;

```

ip\_input.c

The • marks the position of *offset* relative to the address of the outgoing interface is placed in the original route specified  $A_3$  as the first-hop dest recorded in the route. In this way, the route table recorded route should be reversed by the destination packets so that they follow the same path as the source.

Except for UDP, Net/3 reverses a received source route.

583-612

Net/3 sends an ICMP parameter problem error if the offset is smaller than 4 (IPOPT\_MINOFF). If the offset matches one of the local addresses and the option is IPOPT\_LSRR, an ICMP source route failure error is sent. If a local system sent the packet to the wrong host. This means IP must forward the packet.

## End of source route

613-620

Decrementing off converts it to a byte offset from the header. If the header is one of the local addresses and off points to the save\_rte pointer, there are no more addresses in the source route and save\_rte makes a copy of the route in the static table. The addresses in the route in the global ip\_nhops (IPOPT\_ROUTEINFO) table are then freed.

ip\_srcrt is declared as an external static struct declared in ip\_input.c.

## Update packet for next hop

621-637

If ip\_dst is one of the local addresses and offse system is an intermediate system specified in t its final destination. During strict routing, the n network. ifa\_ifwithdst and ifa\_ifwithnet locate a configured interfaces for a matching destination matching network address (a broadcast interface).  
[9.11](#)) locates the route to the next system by c route is found for the next system, an ICMP sof

638-644

If an interface or a route is located, ip\_dooptio off. Within the source route option, the interme the outgoing interface, and the offset is increm

## Multicast destinations

645-646

If the new destination address is not a multicas

the packet should be forwarded after ip\_doopti  
returning the packet to ipintr.

Multicast addresses within a source route enable  
through intermediate routers that don't support  
technique in more detail.

Section 8.5 of Volume 1 contains more examples.

## save\_rte Function

RFC 1122 requires that the route recorded in a  
protocol at the final destination. The transport layer  
to any reply packets. The function save\_rte, sh  
ip\_srcrt structure, shown in [Figure 9.16](#)

**Figure 9.16. ip\_**

```
ip_input.c
57 int      ip_nhops = 0;
58 static struct ip_srcrt {
59     struct in_addr dst;          /* final destination */
60     char    nop;               /* one NOP to align */
61     char    srcopt[IPOPT_OFFSET + 1]; /* OPTVAL, OLEN and OFFSET */
62     struct in_addr route[MAX_IPOPTLEN / sizeof(struct in_addr)];
63 } ip_srcrt;
```

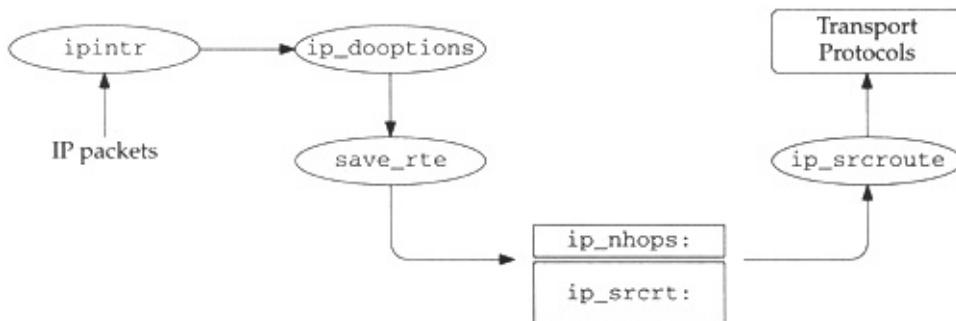
The declaration of route is incorrect, though t

```
struct in_addr route[ (MAX_IPOPT)
```

The discussion with Figures 9.26 and 9.27 covers the code in ip\_input.c. The first part of the code handles IP options, and the second part handles source routes. This section focuses on the source route handling.

This code defines the ip\_srcrt structure and describes how two functions access it: save\_rte, which copies options into ip\_srcrt; and ip\_srcroute, which creates a source route. Figure 9.17 illustrates source route processing.

**Figure 9.17. Processing of IP Options**



**Figure 9.18. save\_rte Function**

```
759 void ip_input.c  
760 save_rte(option, dst)  
761 u_char *option;  
762 struct in_addr dst;  
763 {  
764     unsigned olen;  
765     olen = option[IPOPT_OLEN];  
766     if (olen > sizeof(ip_srcrt) - (1 + sizeof(dst)))  
767         return;  
768     bcopy((caddr_t) option, (caddr_t) ip_srcrt.srcopt, olen);  
769     ip_nhops = (olen - IPOPT_OFFSET - 1) / sizeof(struct in_addr);  
770     ip_srcrt.dst = dst;  
771 }
```

759-771

ip\_dooptions calls save\_rte when a source route option is a pointer to a packet's source route option (i.e., the destination of the return route, HS from the ip\_srcrt structure, save\_rte returns immediate return route information).

This would never happen, as the ip\_srcrt structure is 40 bytes long.

save\_rte copies the option into ip\_srcrt, computes the return route in ip\_nhops, and saves the destination of the return route in ip\_dst.

## ip\_srcroute Function

When responding to a packet, ICMP and the stack call ip\_srcroute to restore the source route that the packet carried. The reverse of ip\_srcroute is ip\_dooptions. Figure 9.19 shows the ip\_srcroute function.

**Figure 9.19. ip\_srcroute Function**

---

```

777 struct mbuf *
778 ip_srcroute()
779 {
780     struct in_addr *p, *q;
781     struct mbuf *m;
782
783     if (ip_nhops == 0)
784         return ((struct mbuf *) 0);
785     m = m_get(M_DONTWAIT, MT_SOOPTS);
786     if (m == 0)
787         return ((struct mbuf *) 0);
788
789 #define OPTSIZ  (sizeof(ip_srcrt.nop) + sizeof(ip_srcrt.srcopt))
790
791     /* length is (nhops+1)*sizeof(addr) + sizeof(nop + srcrt header) */
792     m->m_len = ip_nhops * sizeof(struct in_addr) + sizeof(struct in_addr) +
793                 OPTSIZ;
794
795     /*
796      * First save first hop for return route
797      */
798     p = &ip_srcrt.route[ip_nhops - 1];
799     *(mtod(m, struct in_addr *)) = *p--;
800
801     /*
802      * Copy option fields and padding (nop) to mbuf.
803      */
804     ip_srcrt.nop = IPOPT_NOP;
805     ip_srcrt.srcopt[IPOPT_OFFSET] = IPOPT_MINOFF;
806     bcopy((caddr_t) &ip_srcrt.nop,
807           mtod(m, caddr_t) + sizeof(struct in_addr), OPTSIZ);
808     q = (struct in_addr *) (mtod(m, caddr_t) +
809                           sizeof(struct in_addr) + OPTSIZ);
810
811 #undef OPTSIZ
812
813     /*
814      * Record return path as an IP source route,
815      * reversing the path (pointers are now aligned).
816      */
817     while (p >= ip_srcrt.route) {
818         *q++ = *p--;
819     }
820
821     /*
822      * Last hop goes to final destination.
823      */
824     *q = ip_srcrt.dst;
825     return (m);
826 }

```

---

ip\_input.c

## 777-783

ip\_srcroute reverses the route saved in the ip\_srcrt structure as an ipoption structure ([Figure 9.26](#)). If ip\_nhops is zero, it returns a null pointer.

Recall that in [Figure 8.13](#), ipintr cleared ip\_nhops to zero. All protocols must call ip\_srcroute and save the resulting mbuf when a packet arrives. As noted earlier, this is OK since ip\_srcroute only copies the route information from the ip\_srcrt structure to the mbuf.

by ipintr for each packet, before the next pac

## Allocate mbuf for source route

784-790

If ip\_nhops is nonzero, ip\_srcroute allocates an mbuf for the first-hop destination, the option header info. If the allocation fails, a null pointer is returned as

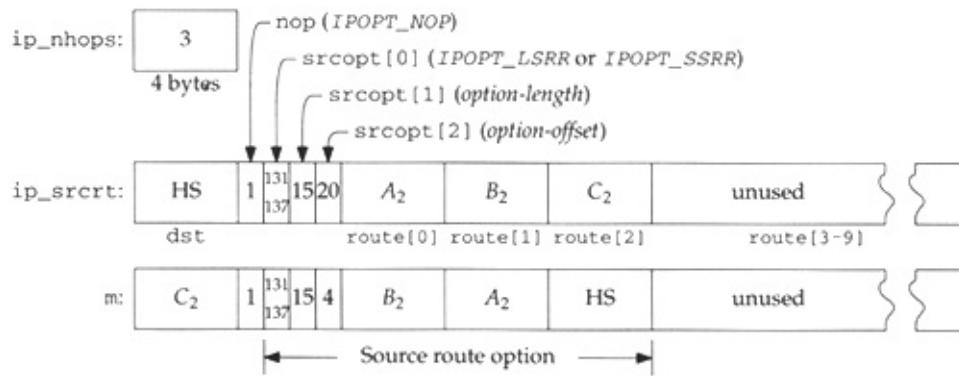
791-804

p is initialized to point to the end of the incoming mbuf. It then records the recorded address to the front of the mbuf where it will be copied for the reversed route. Then the function copies the remaining route information into the mbuf.

805-818

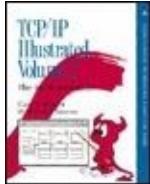
The while loop copies the remaining IP address in reverse order. The last address in the route is saved in the current packet, which save\_rte placed in ip\_srcrt.dst. A diagram illustrates the construction of the reversed route.

Figure 9.20. ip\_srcroute re




---

**Team-Fly**



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.7 Timestamp Option

The timestamp option causes each system to record its notion of the current time within the option as the packet traverses an internet. The time is expected to be in milliseconds since midnight UTC, and is recorded in a 32-bit field.

If the system does not keep accurate UTC (within a few minutes) or the time is not updated at least 15 times per second, it is not considered a standard time. A nonstandard time must have the high-order bit of the timestamp field set.

There are three types of timestamp options, which Net/3 accesses through the

ip\_timestamp structure shown in [Figure 9.22](#).

114-133

As in the ip structure ([Figure 8.10](#)), #ifs ensure that the bit fields access the correct bits in the option. [Figure 9.21](#) lists the three types of timestamp options specified by ipt\_flg.

**Figure 9.21. Possible values for ipt\_flg.**

ipt_flg	Value	Description
<i>IPOPT_TS_TSONLY</i>	0	record timestamps
<i>IPOPT_TS_TSANDADDR</i>	1	record addresses and timestamps
	2	reserved
<i>IPOPT_TS_PRESPEC</i>	3	record timestamps only at the prespecified systems
	4-15	reserved

The originating host must construct the timestamp option with a data area large enough to hold all expected timestamps and addresses. For a timestamp option with an ipt\_flg of 3, the originating host fills in the addresses of the systems at which a time-stamp should be recorded when it constructs the option. [Figure 9.23](#) shows the organization of the three timestamp options.

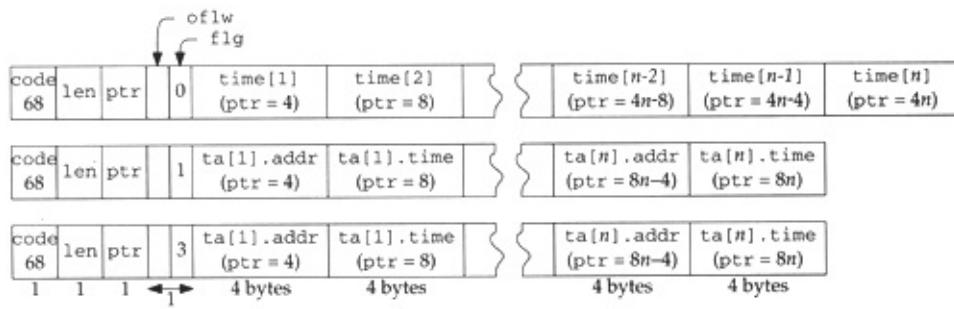
**Figure 9.22. ip\_timestamp structure and constants.**

```

114 struct ip_timestamp {
115     u_char ipt_code;           /* IPOPT_TS */
116     u_char ipt_len;          /* size of structure (variable) */
117     u_char ipt_ptr;          /* index of current entry */
118 #if BYTE_ORDER == LITTLE_ENDIAN
119     u_char ipt_flg:4,        /* flags, see below */
120         ipt_oflw:4;          /* overflow counter */
121 #endif
122 #if BYTE_ORDER == BIG_ENDIAN
123     u_char ipt_oflw:4,        /* overflow counter */
124         ipt_flg:4;          /* flags, see below */
125 #endif
126     union ipt_timestamp {
127         n_long ipt_time[1];
128         struct ipt_ta {
129             struct in_addr ipt_addr;
130             n_long ipt_time;
131         } ipt_ta[1];
132     } ipt_timestamp;
133 };

```

**Figure 9.23. The three timestamp options (ipt omitted).**



Because only 40 bytes are available for IP options, the timestamp options are limited to nine timestamps (ipt\_flg equals 0) or four pairs of addresses and timestamps

(ipt\_flg equals 1 or 3). [Figure 9.24](#) shows the processing for the three different timestamp option types.

**Figure 9.24. ip\_dooptions function:  
timestamp option processing.**

```

674     case IPOPT_TS:
675         code = cp - (u_char *) ip;
676         ipt = (struct ip_timestamp *) cp;
677         if (ipt->ipt_len < 5)
678             goto bad;
679         if (ipt->ipt_ptr > ipt->ipt_len - sizeof(long)) {
680             if (++ipt->ipt_oflw == 0)
681                 goto bad;
682             break;
683         }
684         sin = (struct in_addr *) (cp + ipt->ipt_ptr - 1);
685         switch (ipt->ipt_flg) {
686
687             case IPOPT_TS_TSONLY:
688                 break;
689
690             case IPOPT_TS_TSANDADDR:
691                 if (ipt->ipt_ptr + sizeof(n_time) +
692                     sizeof(struct in_addr) > ipt->ipt_len)
693                     goto bad;
694                 ipaddr.sin_addr = dst;
695                 ia = (INA) ifaof_ifpforaddr((SA) & ipaddr,
696                                              m->m_pkthdr.rcvif);
697                 if (ia == 0)
698                     continue;
699                 bcopy((caddr_t) & IA_SIN(ia)->sin_addr,
700                       (caddr_t) sin, sizeof(struct in_addr));
701                 ipt->ipt_ptr += sizeof(struct in_addr);
702                 break;
703
704             case IPOPT_TS_PRESPEC:
705                 if (ipt->ipt_ptr + sizeof(n_time) +
706                     sizeof(struct in_addr) > ipt->ipt_len)
707                     goto bad;
708                 bcopy((caddr_t) sin, (caddr_t) & ipaddr.sin_addr,
709                       sizeof(struct in_addr));
710                 if (ifa_ifwithaddr((SA) & ipaddr) == 0)
711                     continue;
712                 ipt->ipt_ptr += sizeof(struct in_addr);
713                 break;
714
715             default:
716                 goto bad;
717         }
718     }
719 }
```

ip\_input.c

## 674-684

ip\_dooptions sends an ICMP parameter problem error if the option length is less than 5 bytes (the minimum size of a timestamp option). The oflw field counts the number of systems unable to register

timestamps because the data area of the option was full, oflw is incremented if the data area is full, and when it itself overflows at 16 (it is a 4-bit field), an ICMP parameter problem error is sent.

## Timestamp only

685-687

For a timestamp option with an ipt\_flg of 0 (IPOPT\_TS\_TSONLY), all the work is done after the switch.

## Timestamp and address

688-700

For a timestamp option with an ipt\_flg of 1 (IPOPT\_TS\_TSANDADDR), the address of the receiving interface is recorded (if room remains in the data area), and the option pointer is advanced. Because Net/3 supports multiple IP addresses on a single interface, ip\_dooptions calls ifaof\_ifpforaddr to select the address that best matches the original destination

address of the packet (i.e., the destination before any source routing has occurred). If there is no match, the timestamp option is skipped. (INA and SA were defined in [Figure 9.15](#).)

## Timestamp at prespecified addresses

701-710

If `ipt_flg` is 3 (`IPOPT_TS_PRESPEC`), `ifa_ifwithaddr` determines if the next address specified in the option matches one of the system's addresses. If not, this option requires no processing at this system; the `continue` forces `ip_dooptions` to proceed to the next option. If the next address matches one of the system's addresses, the option pointer is advanced to the next position and control continues after the switch.

## Insert timestamp

711-713

Invalid `ipt_flg` values are caught at default

where control jumps to bad.

714-719

The timestamps are placed in the option by the code that follows the switch statement, iptime returns the number of milliseconds since midnight UTC. ip\_dooptions records the timestamp and increments the option offset to the next position.

## iptime Function

Figure 9.25 shows the implementation of iptime.

### Figure 9.25. iptime function.

```
458 n_time
459 iptime()
460 {
461     struct timeval atv;
462     u_long t;
463     microtime(&atv);
464     t = (atv.tv_sec % (24 * 60 * 60)) * 1000 + atv.tv_usec / 1000;
465     return (htonl(t));
466 }
```

458-466

`microtime` returns the time since midnight January 1, 1970, UTC, in a `timeval` structure. The number of milliseconds since midnight is computed using `atv` and returned in network byte order.

Section 7.4 of Volume 1 provides several timestamp option examples.

---

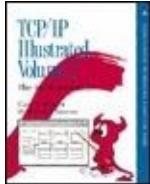
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.8 ip\_insertoptions Function

We saw in [Section 8.6](#) that the `ip_output` function accepts a packet and options. When the function is called from `ip_forward`, the options are already part of the packet so `ip_forward` always passes a null option pointer to `ip_output`. The transport protocols, however, may pass options to `ip_output` where they are merged with the packet by `ip_insertoptions` (called by `ip_output` in [Figure 8.22](#)).

`ip_insertoptions` expects the options to be formatted in an `ipoption` structure, shown in [Figure 9.26](#).

## Figure 9.26. ipoption structure.

```
92 struct ipoption {  
93     struct in_addr ipopt_dst; /* first-hop dst if source routed */  
94     char    ipopt_list[MAX_IPOPTLEN]; /* options proper */  
95 };
```

92-95

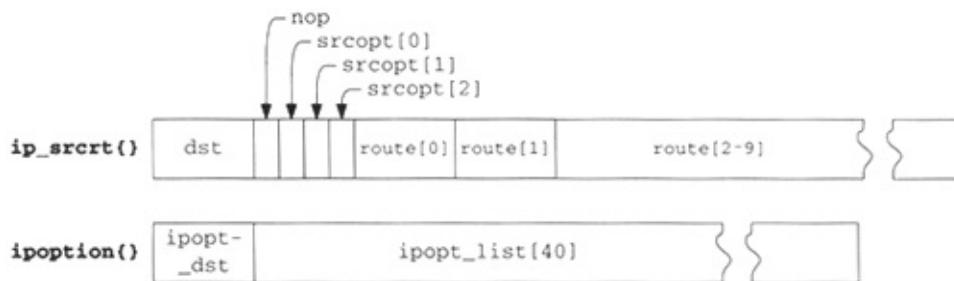
The structure has only two members: ipopt\_dst, which contains the first-hop destination if the option list contains a source route, and ipopt\_list, which is an array of at most 40 (MAX\_IPOPTLEN) bytes of options formatted as we have described in this chapter. If the option list does not include a source route, ipopt\_dst is all 0s.

Note that the ip\_srcrt structure ([Figure 9.16](#)) and the mbuf returned by ip\_srcroute ([Figure 9.19](#)) both conform to the format specified by the ipoption structure. [Figure 9.27](#) compares the ip\_srcrt and ipoption structures.

The ip\_srcrt structure is 4 bytes larger than the ipoption structure. The last entry in the route array (route[9]) is

never filled because it would make the source route option 44 bytes long, larger than the IP header can accommodate ([Figure 9.16](#)).

**Figure 9.27. The ip\_srcrt and ipoption structures.**



The `ip_insertoptions` function is shown in [Figure 9.28](#).

**Figure 9.28. ip\_insertoptions function.**

---

```

352 static struct mbuf *
353 ip_insertoptions(m, opt, phlen)
354 struct mbuf *m;
355 struct mbuf *opt;
356 int      *phlen;
357 {
358     struct ipoption *p = mtod(opt, struct ipoption *);
359     struct mbuf *n;
360     struct ip *ip = mtod(m, struct ip *);
361     unsigned optlen;
362     optlen = opt->m_len - sizeof(p->ipopt_dst);
363     if (optlen + (u_short) ip->ip_len > IP_MAXPACKET)
364         return (m); /* XXX should fail */
365     if (p->ipopt_dst.s_addr)
366         ip->ip_dst = p->ipopt_dst;
367     if (m->m_flags & M_EXT || m->m_data - optlen < m->m_pktdat) {
368         MGETHDR(n, M_DONTWAIT, MT_HEADER);
369         if (n == 0)
370             return (m);
371         n->m_pkthdr.len = m->m_pkthdr.len + optlen;
372         m->m_len -= sizeof(struct ip);
373         m->m_data += sizeof(struct ip);
374         n->m_next = m;
375         m = n;
376         m->m_len = optlen + sizeof(struct ip);
377         m->m_data += max_linkhdr;
378         bcopy((caddr_t) ip, mtod(m, caddr_t), sizeof(struct ip));
379     } else {
380         m->m_data -= optlen;
381         m->m_len += optlen;
382         m->m_pkthdr.len += optlen;
383         ovbcopy((caddr_t) ip, mtod(m, caddr_t), sizeof(struct ip));
384     }
385     ip = mtod(m, struct ip *);
386     bcopy((caddr_t) p->ipopt_list, (caddr_t) (ip + 1), (unsigned) optlen);
387     *phlen = sizeof(struct ip) + optlen;
388     ip->ip_len += optlen;
389     return (m);
390 }

```

---

ip\_output.c

## 352-364

ip\_insertoptions has three arguments: m, the outgoing packet; opt, the options formatted in an ipoption structure; and phlen, a pointer to an integer where the new header length (after options are inserted) is returned. If the size of packet with the options exceeds the maximum packet size of 65,535 (IP\_MAXPACKET) bytes, the options are silently discarded.

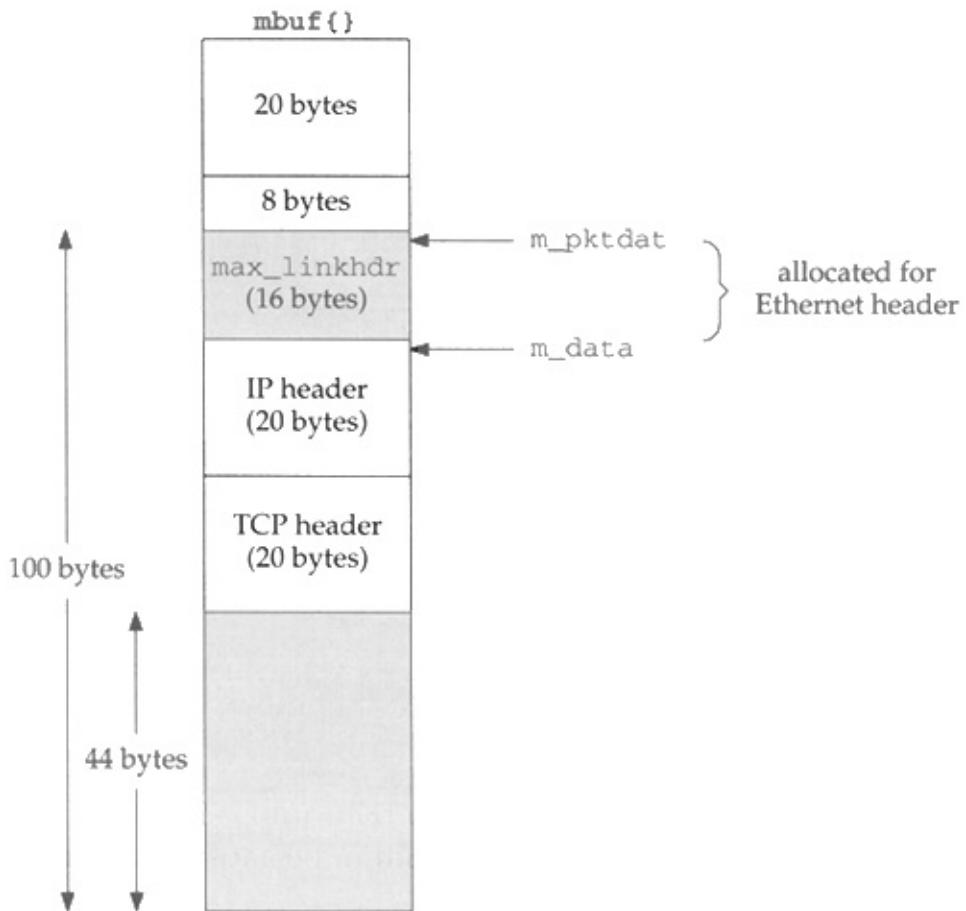
`ip_output` does not expect `ip_insertoptions` ever to fail, so there is no way to report the error. Fortunately, few applications attempt to send a maximally sized datagram, let alone one with options.

365-366

If `ipopt_dst.s_addr` specifies a nonzero address, then the options include a source route and `ip_dst` in the packet's header is replaced with the first-hop destination from the source route.

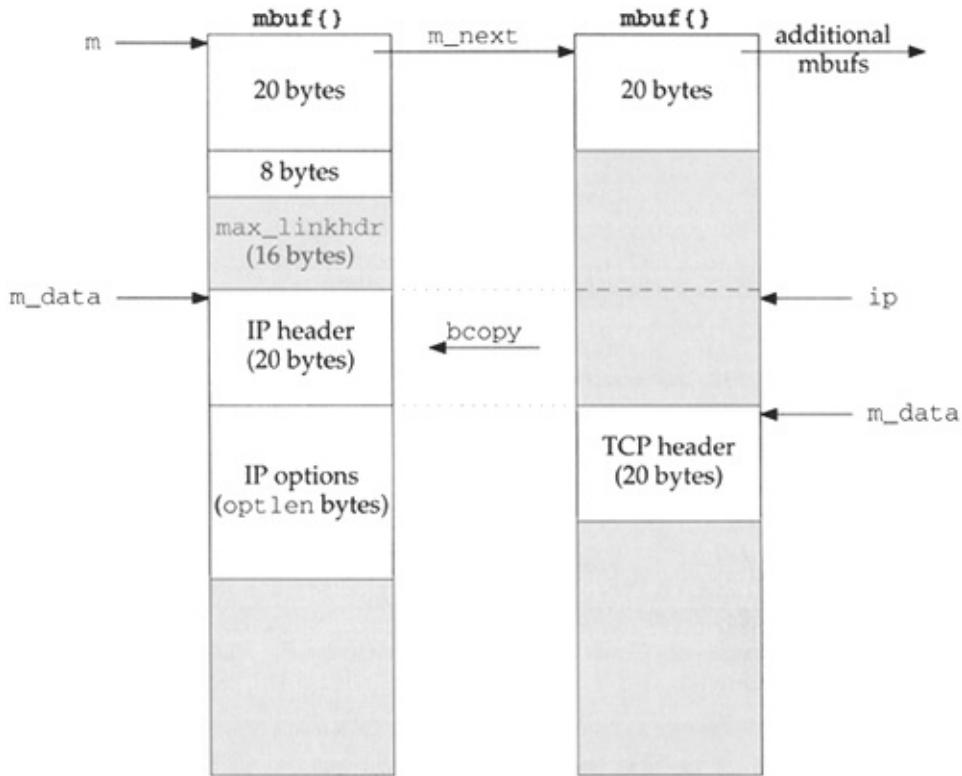
In [Section 26.2](#) we'll see that TCP calls `MGETHDR` to allocate a separate mbuf for the IP and TCP headers. [Figure 9.29](#) shows the mbuf organization for a TCP segment before the code in lines 367 to 378 is executed.

**Figure 9.29. `ip_insertoptions` function:  
TCP segment.**



If the options to be inserted occupy more than 16 bytes, the test on line 367 is true and MGETHDR is called to allocate an additional mbuf. [Figure 9.30](#) shows the organization of the mbufs after the options have been copied into the new mbuf.

**Figure 9.30. ip\_insertoptions function:  
TCP segment, after options have been  
copied.**

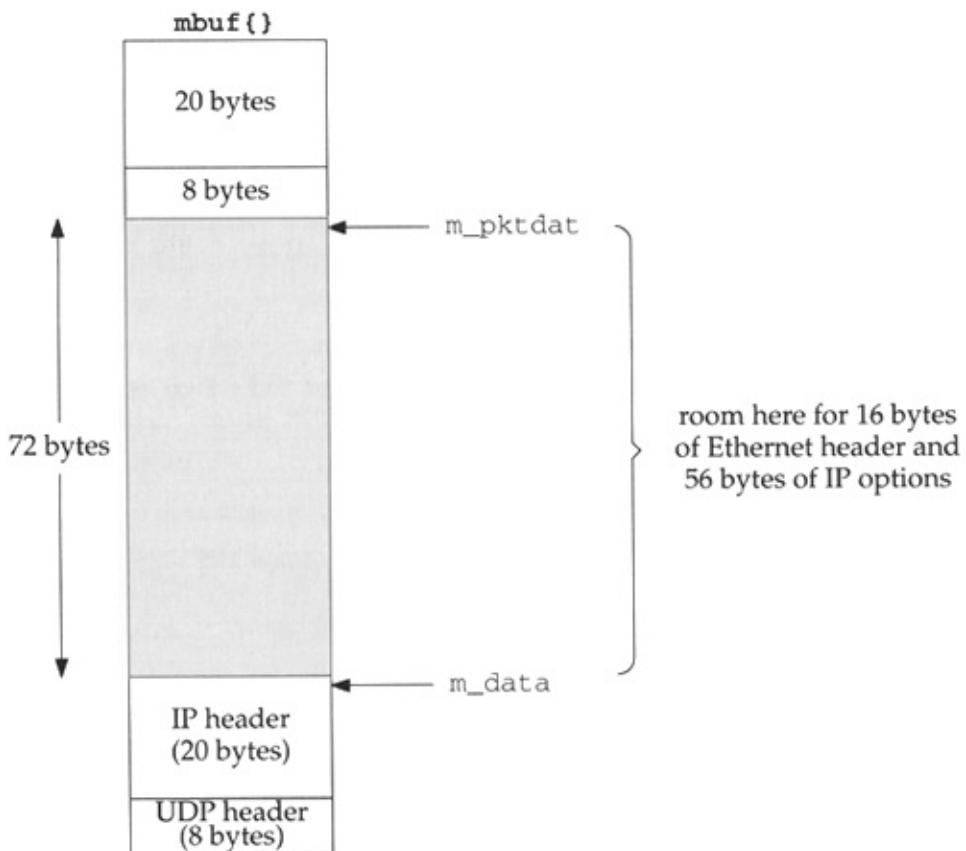


367-378

If the packet header is stored in a cluster, or the first mbuf does not have room for the options, `ip_insertoptions` allocates a new packet header mbuf, initializes its length, trims the IP header from the old mbuf, and moves the header from the old mbuf to the new mbuf.

As described in [Section 23.6](#), UDP uses `M_PREPEND` to place the UDP and IP headers at the end of an mbuf, separate from the data. This is illustrated in [Figure 9.31](#).

**Figure 9.31. ip\_insertoptions function:  
UDP datagram.**



Because the headers are located at the end of the mbuf, there is always room for IP options in the mbuf and the condition on line 367 is always false for UDP.

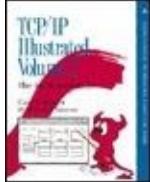
379-384

If the packet has room at the beginning of the mbuf's data area for the options,

`m_data` and `m_len` are adjusted to contain `optlen` more bytes, and the current IP header is moved by `ovbcopy` (which can handle overlapping source and destinations) to leave room for the options.

**385-390**

`ip_insertoptions` can now copy the `ipopt_list` member of the `ipoption` structure directly into the mbuf just after the IP header. `ip_insertoptions` stores the new header length in `*phlen`, adjusts the datagram length (`ip_len`), and returns a pointer to the packet header mbuf.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.9 ip\_pcbopts Function

The `ip_pcbopts` function converts the list of IP options provided with the `IP_OPTIONS` socket option into the form expected by `ip_output`: an `ipoption` structure.

**Figure 9.32. `ip_pcbopts` function.**

```
559 int
560 ip_pcbopts(pcbopt, m)
561 struct mbuf **pcbopt;
562 struct mbuf *m;
563 {
564     int cnt, optlen;
565     u_char *cp;
566     u_char opt;
567     /* turn off any old options */
568     if (*pcbopt)
569         (void) m_free(*pcbopt);
570     *pcbopt = 0;
571     if (m == (struct mbuf *) 0 || m->m_len == 0) {
572         /*
573          * Only turning off any previous options.
574          */
575         if (m)
576             (void) m_free(m);
577         return (0);
578     }
579     if (m->m_len % sizeof(long))
580         goto bad;
581     /*
582      * IP first-hop destination address will be stored before
583      * actual options; move other options back
584      * and clear it when none present.
585      */
586     if (m->m_data + m->m_len + sizeof(struct in_addr) >= &m->m_dat[MLEN])
587         goto bad;
588     cnt = m->m_len;
589     m->m_len += sizeof(struct in_addr);
590     cp = mtod(m, u_char *) + sizeof(struct in_addr);
591     ovbcopy(mtod(m, caddr_t), (caddr_t) cp, (unsigned) cnt);
592     bzero(mtod(m, caddr_t), sizeof(struct in_addr));
593     for (; cnt > 0; cnt -= optlen, cp += optlen) {
594         opt = cp[IPOPT_OPTVAL];
595         if (opt == IPOPT_EOL)
596             break;
597         if (opt == IPOPT_NOP)
598             optlen = 1;
599         else {
600             optlen = cp[IPOPT_OLEN];
601             if (optlen <= IPOPT_OLEN || optlen > cnt)
602                 goto bad;
603         }
604     }
```

```

604     switch (opt) {
605     default:
606         break;
607     case IPOPT_LSRR:
608     case IPOPT_SSRR:
609     /*
610      * user process specifies route as:
611      * ->A->B->C->D
612      * D must be our final destination (but we can't
613      * check that since we may not have connected yet).
614      * A is first hop destination, which doesn't appear in
615      * actual IP option, but is stored before the options.
616      */
617     if (optlen < IPOPT_MINOFF - 1 + sizeof(struct in_addr))
618         goto bad;
619     m->m_len -= sizeof(struct in_addr);
620     cnt -= sizeof(struct in_addr);
621     optlen -= sizeof(struct in_addr);
622     cp[IPOPT_OLEN] = optlen;
623     /*
624      * Move first hop before start of options.
625      */
626     bcopy((caddr_t) & cp[IPOPT_OFFSET + 1], mtod(m, caddr_t),
627           sizeof(struct in_addr));
628     /*
629      * Then copy rest of options back
630      * to close up the deleted entry.
631      */
632     ovbcopy((caddr_t) (&cp[IPOPT_OFFSET + 1] +
633                      sizeof(struct in_addr)),
634            (caddr_t) & cp[IPOPT_OFFSET + 1],
635            (unsigned) cnt + sizeof(struct in_addr));
636     break;
637   }
638 }
639 if (m->m_len > MAX_IPOPTLEN + sizeof(struct in_addr))
640   goto bad;
641 *pcbopt = m;
642 return (0);
643 bad:
644   (void) m_free(m);
645   return (EINVAL);
646 }

```

ip\_output.c

## 559-562

The first argument, pcbopt, references the pointer to the current list of options. The function replaces this pointer with a pointer to the new list of options constructed from options specified in the mbuf chain pointed to by the second argument, m. The option list prepared by

the process to be included with the IP\_OPTIONS socket option looks like a standard list of IP options except for the format of the LSRR and SSRR options. For these options, the first-hop destination is included as the first address in the route.

[Figure 9.14](#) shows that the first-hop destination appears as the destination address in the outgoing packet, not as the first address in the route.

## Discard previous options

563-580

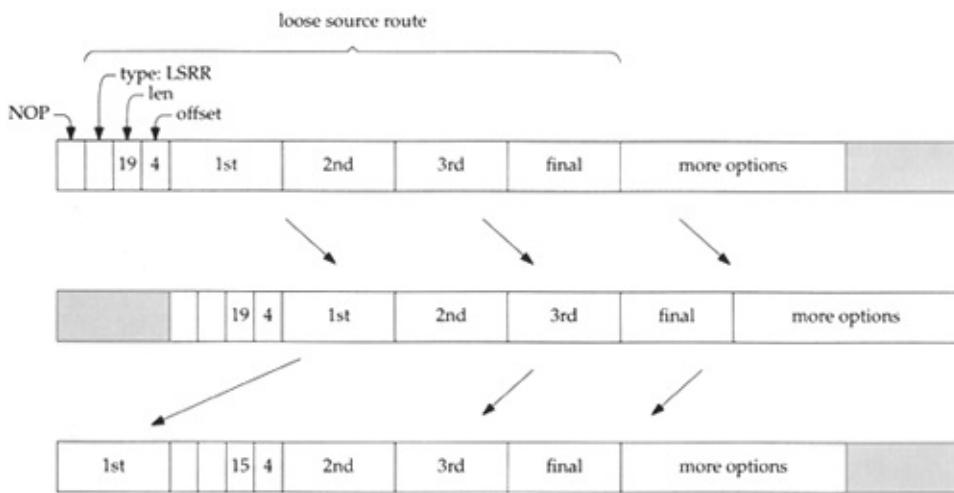
Any previous options are discarded by m\_free and \*pcbopt is cleared. If the process passed an empty mbuf or didn't pass an mbuf at all, the function returns immediately without installing any new options.

If the new list of options is not padded to a 4-byte boundary, ip\_pcbopts jumps to bad, discards the list and returns EINVAL.

The remainder of the function rearranges

the list to look like an ipoption structure. Figure 9.33 illustrates this process.

**Figure 9.33. ip\_pcbopts option list processing.**



## Make room for first-hop destination

581-592

If there is room in the mbuf, all the data is shifted by 4 bytes (the size of an in\_addr structure) toward the end of the mbuf. ovbcopy performs the copy. bzero clears the 4 bytes at the start of the mbuf.

## Scan option list

593-606

The for loop scans the option list looking for LSRR and SSRR options. For multibyte options, the loop also verifies that the length of the option is reasonable.

## Rearrange LSRR or SSRR option

607-638

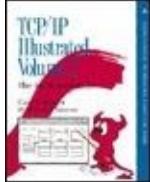
When the loop locates a LSRR or SSRR option, it decrements the mbuf size, the loop index, and the option length by 4, since the first address in the option will be removed and shifted to the front of the mbuf.

bcopy moves the first address and ovbcopy shifts the remainder of the options by 4 bytes to fill the gap left by the first address.

## Cleanup

## 639-646

After the loop, the size of the option list (including the first-hop address) must be no more than 44 (MAX\_IPOPTLEN+4) bytes. A larger list does not fit in the IP packet header. The list is saved in \*pcbopt and the function returns.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.10 Limitations

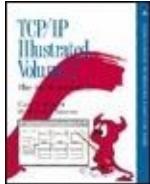
Options are rarely present in IP datagrams other than those created by administrative and diagnostic tools. Volume 1 discusses two of the more common tools, ping and traceroute. It is difficult to write applications that utilize IP options. The programming interfaces are poorly documented and not well standardized. Most vendor supplied applications, such as Telnet and FTP, do not provide a way for a user to specify options such as a source route.

The usefulness of the record route, timestamp, and source route options in a large internet is limited by the maximum

size of an IP header. Most routes contain more hops than can be represented in the 40 option bytes. When multiple options appear in the same packet, the available space is almost useless. IPv6 addresses this problem with a more flexible option header design.

During fragmentation, IP copies only some options into the noninitial fragments, since the options in noninitial fragments are discarded during reassembly. Only options from the initial fragment are made available to the transport protocol at the destination ([Section 10.6](#)). But some, such as source route, must be copied to each fragment, even if they are discarded in noninitial fragments at the destination.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 9. IP Option Processing

### 9.11 Summary

In this chapter we showed the format and processing of IP options. We didn't cover the security and stream ID options since they are not implemented in Net/3.

We saw that the size of multibyte options is fixed by the source host when it constructs the option. The usefulness of IP options is severely limited by the small maximum option header size of 40 bytes.

The source route options require the most support. Incoming source routes are saved by `save_rte` and reversed by `ip_srcroute`. A host that does not normally forward packets may forward source routed

packets, but RFC 1122 requires this capability to be disabled by default. Net/3 does not have a switch for this feature and always forwards source routed packets.

Finally, we saw how options are merged into an outgoing packet by `ip_insertoptions`.

## Exercises

What would happen if a packet  
**9.1** contained two different source route options?

Some commercial routers can be configured to discard packets based on their IP destination address. In this way, a machine or group of machines can be isolated from the larger internet beyond the router. Describe  
**9.2** how source routed packets can bypass this mechanism. Assume that there is at least one host within the network that the router is not blocking, and

that it forwards source routed datagrams.

Some hosts may not be configured with a default route. In general, this prevents communication with the host

**9.3** since the host can't route to destinations outside its directly connected networks. Describe how a source route can enable communication with this type of host.

**9.4** Why is a NOP used in the ip\_srcrt structure in [Figure 9.16](#)?

Can a nonstandard time value be confused with a standard time value in the timestamp options?

**9.6** ip\_dooptions saves the destination address of the packet in dest before processing any options ([Figure 9.8](#)). Why?

---

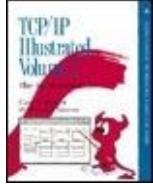
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 10. IP Fragmentation and Reassembly

[Section 10.1. Introduction](#)

[Section 10.2. Code Introduction](#)

[Section 10.3. Fragmentation](#)

[Section 10.4. ip\\_optcopy Function](#)

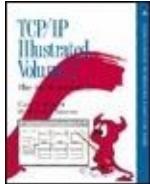
[Section 10.5. Reassembly](#)

[Section 10.6. ip\\_reass Function](#)

[Section 10.7. ip\\_slowtimo Function](#)

[Section 10.8. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.1 Introduction

In this chapter we describe the IP fragmentation and reassembly processing that we postponed in [Chapter 8](#).

IP has an important capability of being able to fragment a packet when it is too large to be transmitted by the selected hardware interface. The oversized packet is split into two or more IP fragments, each of which is small enough to be transmitted on the selected network. Fragments may be further split by routers farther along the path to the final destination. Thus, at the destination host, an IP datagram can be contained in a

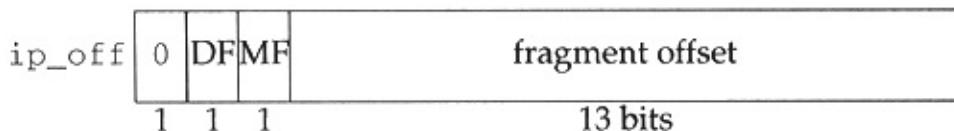
single IP packet or, if it was fragmented in transit, it can arrive in multiple IP packets. Because individual fragments may take different paths to the destination host, only the destination host has a chance to see all the fragments. Thus only the destination host can reassemble the fragments into a complete datagram to be delivered to the appropriate transport protocol.

[Figure 8.5](#) shows that 0.3% ( $72, 786/27, 881, 978$ ) of the packets received were fragments and 0.12% ( $260, 484/(29, 447, 726796, 084)$ ) of the datagrams sent were fragmented. On world.std.com, 9.5% of the packets received were fragments. world has more NFS activity, which is a common source of IP fragmentation.

Three fields in the IP header implement fragmentation and reassembly: the identification field (ip\_id), the flags field (the 3 high-order bits of ip\_off), and the offset field (the 13 low-order bits of ip\_off). The flags field is composed of three 1-bit flags. Bit 0 is reserved and must be 0, bit 1 is the "don't fragment"

(DF) flag, and bit 2 is the "more fragments" (MF) flag. In Net/3, the flag and offset fields are combined and accessed by `ip_off`, as shown in [Figure 10.1](#).

**Figure 10.1. `ip_off` controls fragmentation of an IP packet.**



Net/3 accesses the DF and MF bits by masking `ip_off` with `IP_DF` and `IP_MF` respectively. An IP implementation must allow an application to request that the DF bit be set in an outgoing datagram.

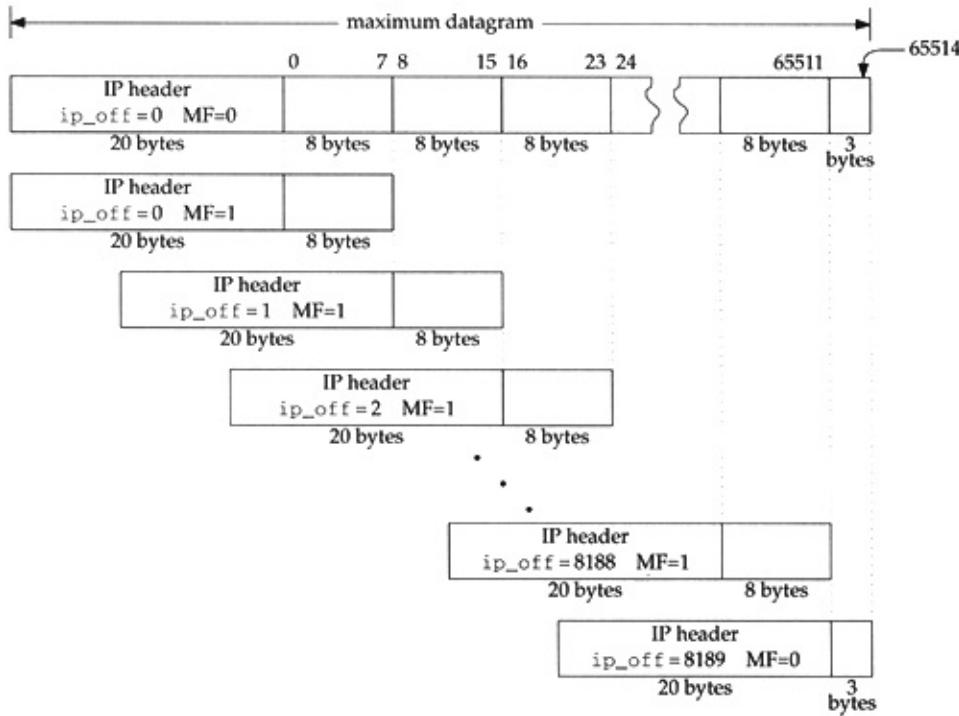
Net/3 does not provide *application-level* control over the DF bit when using UDP or TCP.

A process may construct and send its own IP headers with the raw IP interface ([Chapter 32](#)). The DF bit may be set by the transport layers directly such as when TCP performs *path MTU*

*discovery.*

The remaining 13 bits of ip\_off specify the fragment's position within the original datagram, measured in 8-byte units. Accordingly, every fragment except the last must contain a multiple of 8 bytes of data so that the following fragment starts on an 8-byte boundary. [Figure 10.2](#) illustrates the relationship between the byte offset within the original datagram and the fragment offset (low-order 13 bits of ip\_off) in the fragment's IP header.

**Figure 10.2. Fragmentation of a 65535-byte datagram.**



**Figure 10.2** shows a maximally sized IP datagram divided into 8190 fragments. Each fragment contains 8 bytes except the last, which contains only 3 bytes. We also show the MF bit set in all the fragments except the last. This is an unrealistic example, but it illustrates several implementation issues.

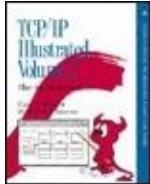
The numbers above the original datagram are the byte offsets for the *data* portion of the datagram. The fragment offset (*ip\_off*) is computed from the start of the data portion of the datagram. It is impossible for a fragment to include a byte beyond

offset 65514 since the reassembled datagram would be larger than 65535 bytes—the maximum value of the ip\_len field. This restricts the maximum value of ip\_off to 8189 ( $8189 \times 8 = 65512$ ), which leaves room for 3 bytes in the last fragment. If IP options are present, the offset must be smaller still.

Because an IP internet is connectionless, fragments from one datagram may be interleaved with those from another at the destination. ip\_id uniquely identifies the fragments of a particular datagram. The source system sets ip\_id in each datagram to a unique value for all datagrams using the same source (ip\_src), destination (ip\_dst), and protocol (ip\_p) values for the lifetime of the datagram on the internet.

To summarize, ip\_id identifies the fragments of a particular datagram, ip\_off positions the fragment within the original datagram, and the MF bit marks every fragment except the last.

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.2 Code Introduction

The reassembly data structures appear in a single header. Reassembly and fragmentation processing is found in two C files. The three files are listed in [Figure 10.3](#).

### Figure 10.3. Files discussed in this chapter.

File	Description
netinet/ip_var.h	reassembly data structures
netinet/ip_output.c	fragmentation code
netinet/ip_input.c	reassembly code

## Global Variables

Only one global variable, ipq, is described in this chapter.

**Figure 10.4. Global variable introduced in this chapter.**

Variable	Type	Description
ipq	struct ipq *	reassembly list

## Statistics

The statistics modified by the fragmentation and reassembly code are shown in [Figure 10.5](#). They are a subset of the statistics included in the ipstat structure described by [Figure 8.4](#).

**Figure 10.5. Statistics collected in this chapter.**

ipstat member	Description
ips_cantfrag	#datagrams not sent because fragmentation was required but was prohibited by the DF bit
ips_odropped	#output packets dropped because of a memory shortage
ips_ofragments	#fragments transmitted
ips_fragmented	#packets fragmented for output

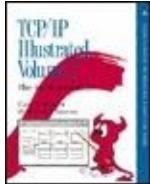
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.3 Fragmentation

We now return to `ip_output` and describe the fragmentation code. Recall from [Figure 8.25](#) that if a packet fits within the MTU of the selected outgoing interface, it is transmitted in a single link-level frame. Otherwise the packet must be fragmented and transmitted in multiple frames. A packet may be a complete datagram or it may itself be a fragment that was created by a previous system. We describe the fragmentation code in three parts:

- determine fragment size ([Figure 10.6](#)),

## Figure 10.6. ip\_output function: determine fragment size.

```
253  /*
254   * Too large for interface; fragment if possible.
255   * Must be able to put at least 8 bytes per fragment.
256   */
257  if (ip->ip_off & IP_DF) {
258      error = EMSGSIZE;
259      ipstat.ips_cantfrag++;
260      goto bad;
261  }
262  len = (ifp->if_mtu - hlen) & ~7;
263  if (len < 8) {
264      error = EMSGSIZE;
265      goto bad;
266 }
```

- construct fragment list (Figure 10.7),  
and

## Figure 10.7. ip\_output function: construct fragment list.

---

```

267  {
268      int      mhlen, firstlen = len;
269      struct mbuf **mnext = &m->m_nextpkt;
270
271      /*
272       * Loop through length of segment after first fragment,
273       * make new header and copy data of each part and link onto chain.
274       */
275      m0 = m;
276      mhlen = sizeof(struct ip);
277      for (off = hlen + len; off < (u_short) ip->ip_len; off += len) {
278          MGETHDR(m, M_DONTWAIT, MT_HEADER);
279          if (m == 0) {
280              error = ENOBUFS;
281              ipstat.ips_odropped++;
282              goto sendorfree;
283          }
284          m->m_data += max_linkhdr;
285          mhip = mtod(m, struct ip *);
286          *mhip = *ip;
287          if (hlen > sizeof(struct ip)) {
288              mhlen = ip_optcopy(ip, mhip) + sizeof(struct ip);
289              mhip->ip_hl = mhlen >> 2;
290          }
291          m->m_len = mhlen;
292          mhip->ip_off = ((off - hlen) >> 3) + (ip->ip_off & ~IP_MF);
293          if (ip->ip_off & IP_MF)
294              mhip->ip_off |= IP_MF;
295          if (off + len >= (u_short) ip->ip_len)
296              len = (u_short) ip->ip_len - off;
297          else
298              mhip->ip_off |= IP_MF;
299          mhip->ip_len = htons((u_short) (len + mhlen));
300          m->m_next = m_copy(m0, off, len);
301          if (m->m_next == 0) {
302              (void) m_free(m);
303              error = ENOBUFS; /* ??? */
304              ipstat.ips_odropped++;
305              goto sendorfree;
306          }
307          m->m_pkthdr.len = mhlen + len;
308          m->m_pkthdr.rcvif = (struct ifnet *) 0;
309          mhip->ip_off = htons((u_short) mhip->ip_off);
310          mhip->ip_sum = 0;
311          mhip->ip_sum = in_cksum(m, mhlen);
312          *mnext = m;
313          mnext = &m->m_nextpkt;
314          ipstat.ips_ofragments++;
}

```

---

- construct initial fragment and send fragments ([Figure 10.8](#)).

## Figure 10.8. ip\_output function: send fragments.

```

315     /*
316      * Update first fragment by trimming what's been copied out
317      * and updating header, then send each fragment (in order).
318      */
319     m = m0;
320     m_adj(m, hlen + firstlen - (u_short) ip->ip_len);
321     m->m_pkthdr.len = hlen + firstlen;
322     ip->ip_len = htons((u_short) m->m_pkthdr.len);
323     ip->ip_off = htons((u_short) (ip->ip_off | IP_MF));
324     ip->ip_sum = 0;
325     ip->ip_sum = in_cksum(m, hlen);
326     sendorfree:
327     for (m = m0; m; m = m0) {
328         m0 = m->m_nextpkt;
329         m->m_nextpkt = 0;
330         if (error == 0)
331             error = (*ifp->if_output) (ifp, m,
332                                         (struct sockaddr *) dst, ro->ro_rt);
333         else
334             m_freem(m);
335     }
336     if (error == 0)
337         ipstat.ips_fragmented++;
338 }
```

ip\_output.c

## 253-261

The fragmentation algorithm is straightforward, but the implementation is complicated by the manipulation of the mbuf structures and chains. If fragmentation is prohibited by the DF bit, ip\_output discards the packet and returns EMSGSIZE. If the datagram was generated on this host, a transport protocol passes the error back to the process, but if the datagram is being forwarded, ip\_forward generates an ICMP destination unreachable error with an indication that the packet could not be forwarded without fragmentation ([Figure 8.21](#)).

Net/3 does not implement the path MTU discovery algorithms used to probe the path to a destination and discover the largest transmission unit supported by all the intervening networks. Sections 11.8 and 24.2 of Volume 1 describe path MTU discovery for UDP and TCP.

262-266

len, the number of data bytes in each fragment, is computed as the MTU of the interface less the size of the packet's header and then rounded down to an 8-byte boundary by clearing the low-order 3 bits ( $\& \sim 7$ ). If the MTU is so small that each fragment contains less than 8 bytes, ip\_output returns EMSGSIZE.

Each new fragment contains an IP header, some of the options from the original packet, and at most len data bytes.

The code in [Figure 10.7](#), which is the start of a C compound statement, constructs the list of fragments starting with the second fragment. The original packet is converted into the initial fragment after

the list is created ([Figure 10.8](#)).

267-269

The extra block allows `mhlen`, `firstlen`, and `mnext` to be declared closer to their use in the function. These variables are in scope until the end of the block and hide any similarly named variables outside the block.

270-276

Since the original mbuf chain becomes the first fragment, the for loop starts with the offset of the second fragment: `hlen + len`. For each fragment `ip_output` takes the following actions:

- 277-284

Allocate a new packet mbuf and adjust its `m_data` pointer to leave room for a 16-byte link-layer header (`max_linkhdr`). If `ip_output` didn't do this, the network interface driver would have to allocate an additional mbuf to hold the link header or move the data. Both are time-consuming tasks that are

easily avoided here.

- 285-290

Copy the IP header and IP options from the original packet into the new packet. The former is copied with a structure assignment. `ip_optcopy` copies only those options that get copied into each fragment ([Section 10.4](#)).

- 291-297

Set the offset field (`ip_off`) for the fragment including the MF bit. If MF is set in the original packet, then MF is set in all the fragments. If MF is not set in the original packet, then MF is set for every fragment except the last.

- 298

Set the length of this fragment accounting for a shorter header (`ip_optcopy` may not have copied all the options) and a shorter data area for the last fragment. The length is stored in network byte order.

- 299-305

Copy the data from the original packet into this fragment. `m_copy` allocates additional mbufs if necessary. If `m_copy` fails, ENOBUFS is posted. Any mbufs already allocated are discarded at sendorfree.

- 306-314

Adjust the mbuf packet header of the newly created fragment to have the correct total length, clear the new fragment's interface pointer, convert `ip_off` to network byte order, compute the checksum for the new fragment, and link the fragment to the previous fragment through `m_nextpkt`.

In [Figure 10.8](#), `ip_output` constructs the initial fragment and then passes each fragment to the interface layer.

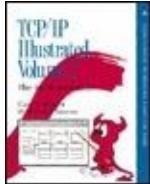
## 315-325

The original packet is converted into the first fragment by trimming the extra data from its end, setting the MF bit, converting

ip\_len and ip\_off to network byte order, and computing the new checksum. All the IP options are retained in this fragment. At the destination host, only the IP options from the first fragment of a datagram are retained when the datagram is reassembled ([Figure 10.28](#)). Some options, such as source routing, must be copied into each fragment even though the option is discarded during reassembly.

326-338

At this point, ip\_output has either a complete list of fragments or an error has occurred and the partial list of fragments must be discarded. The for loop traverses the list either sending or discarding fragments according to error. Any error encountered while sending fragments causes the remaining fragments to be discarded.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.4 ip\_optcopy Function

During fragmentation, `ip_optcopy` (Figure 10.9) copies the options from the incoming packet (if the packet is being forwarded) or from the original datagram (if the datagram is locally generated) into the outgoing fragments.

**Figure 10.9. `ip_optcopy` function.**

```
395 int ip_optcopy(ip, jp)
396 struct ip *ip, *jp;
398 {
399     u_char *cp, *dp;
400     int opt, optlen, cnt;
401     cp = (u_char *) (ip + 1);
402     dp = (u_char *) (jp + 1);
403     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
404     for (; cnt > 0; cnt -= optlen, cp += optlen) {
405         opt = cp[0];
406         if (opt == IPOPT_EOL)
407             break;
408         if (opt == IPOPT_NOP) {
409             /* Preserve for IP mcast tunnel's LSRR alignment. */
410             *dp++ = IPOPT_NOP;
411             optlen = 1;
412             continue;
413         } else
414             optlen = cp[IPOPT_OLEN];
415         /* bogus lengths should have been caught by ip_dooptions */
416         if (optlen > cnt)
417             optlen = cnt;
418         if (IPOPT_COPIED(opt)) {
419             bcopy((caddr_t) cp, (caddr_t) dp, (unsigned) optlen);
420             dp += optlen;
421         }
422     }
423     for (optlen = dp - (u_char *) (jp + 1); optlen & 0x3; optlen++)
424         *dp++ = IPOPT_EOL;
425     return (optlen);
426 }
```

ip\_output.c

## 395-422

The arguments to `ip_optcopy` are: `ip`, a pointer to the IP header of the outgoing packet; and `jp`, a pointer to the IP header of the newly created fragment. `ip_optcopy` initializes `cp` and `dp` to point to the first option byte in each packet and advances `cp` and `dp` as it processes each option. The first for loop copies a single option during each iteration stopping when it encounters an EOL option or when it has examined all the options. NOP options are copied to

preserve any alignment constraints in the subsequent options.

The Net/2 release discarded NOP options.

If IPOPT\_COPIED indicates that the *copied* bit is on, ip\_optcopy copies the option to the new fragment. [Figure 9.5](#) shows which options have the *copied* bit set. If an option length is too large, it is truncated; ip\_dooptions should have already discovered this type of error.

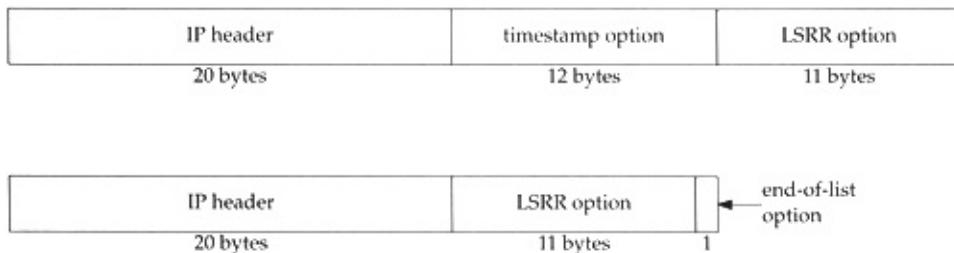
423-426

The second for loop pads the option list out to a 4-byte boundary. This is required, since the packet's header length (ip\_hlen) is measured in 4-byte units. It also ensures that the transport header that follows is aligned on a 4-byte boundary. This improves performance since many transport protocols are designed so that 32-bit header fields are aligned on 32-bit boundaries if the transport header starts on a 32-bit boundary. This arrangement increases performance on CPUs that have

difficulty accessing unaligned 32-bit words.

Figure 10.10 illustrates the operation of ip\_optcopy.

## Figure 10.10. Not all options are copied during fragmentation.



In Figure 10.10 we see that ip\_optcopy does not copy the timestamp option (its *copied* bit is 0) but does copy the LSRR option (its *copied* bit is 1). ip\_optcopy has also added a single EOL option to pad the new options to a 4-byte boundary.

## Chapter 10. IP Fragmentation and Reas

### 10.5 Reassembly

Now that we have described the fragmentation fragment), we return to ipintr and the reassembly code from ipin 8.15 we omitted the reassembly code from ipin discussion. ipintr can pass only entire datagram layer for processing. Fragments that are received to ip\_reass, which attempts to reassemble fragments into datagrams. The code from ipintr is shown in Fig

**Figure 10.11.** ipintr function: fragme

```
271     ours:  
272     /*  
273      * If offset or IP_MF are set, must reassemble.  
274      * Otherwise, nothing need be done.  
275      * (We could look in the reassembly queue to see  
276      * if the packet was previously fragmented,  
277      * but it's not worth the time; just let them time out.)  
278      */  
279     if (ip->ip_off & ~IP_DF) {  
280         if (m->m_flags & M_EXT) { /* XXX */  
281             if ((m = m_pullup(m, sizeof(struct ip))) == 0) {  
282                 ipstat.ips_toosmall++;  
283                 goto next;  
284             }  
285             ip = mtod(m, struct ip *);  
286         }  
287         /*  
288          * Look for queue of fragments  
289          * of this datagram.  
290          */  
291         for (fp = ipq.next; fp != &ipq; fp = fp->next)  
292             if (ip->ip_id == fp->ipq_id &&  
293                 ip->ip_src.s_addr == fp->ipq_src.s_addr &&  
294                 ip->ip_dst.s_addr == fp->ipq_dst.s_addr &&  
295                 ip->ip_p == fp->ipq_p)  
296                 goto found;  
297         fp = 0;  
298     found:  
299         /*  
300          * Adjust ip_len to not reflect header,  
301          * set ip_mff if more fragments are expected,  
302          * convert offset of this to bytes.  
303          */  
304         ip->ip_len -= hlen;  
305         ((struct ipasfrag *) ip)->ipf_mff &= ~1;  
306         if (ip->ip_off & IP_MF)  
307             ((struct ipasfrag *) ip)->ipf_mff |= 1;  
308         ip->ip_off <= 3;  
309         /*  
310          * If datagram marked as having more fragments  
311          * or if this is not the first fragment,  
312          * attempt reassembly; if it succeeds, proceed.  
313          */  
314         if (((struct ipasfrag *) ip)->ipf_mff & 1 || ip->ip_off) {  
315             ipstat.ips_fragments++;  
316             ip = ip_reass((struct ipasfrag *) ip, fp);  
317             if (ip == 0)  
318                 goto next;  
319             ipstat.ips_reassembled++;  
320             m = dtom(ip);  
321         } else if (fp)  
322             ip_freef(fp);  
323     } else  
324         ip->ip_len -= hlen;
```

271-279

Recall that ip\_off contains the DF bit, the MF bi

offset. The DF bit is masked out and if either the offset is nonzero, the packet is a fragment that. If both are zero, the packet is a complete datagram. The code is skipped and the else clause at the end is executed, which excludes the header length from the length.

280-286

m\_pullup moves data in an external cluster into a mbuf. Recall that the SLIP interface ([Section 5](#)). IP packet in an external cluster if it does not fit into a mbuf. m\_devget can return the entire packet in a cluster. Before the dtom macro will work ([Section 2.6](#)), the IP header from the cluster into the data area.

287-297

Net/3 keeps incomplete datagrams on the global list. The name is somewhat confusing since the data is in a queue. That is, insertions and deletions can occur not just at the ends. We'll use the term *list* to emphasize this.

ipintr performs a linear search of the list to locate a datagram for the current fragment. Remember that each fragment is uniquely identified by the 4-tuple: {ip\_id, ip\_src, ip\_dst, port}. An entry in ipq is a list of fragments and fp points to the first fragment. ipintr finds a match.

Net/3 uses linear searches to access many of  
While simple, this method can become a bottleneck  
supporting large numbers of network connections.

## 298-303

At found, the packet is modified by ipintr to facilitate:

- 304

ipintr changes ip\_len to exclude the standard options. We must keep this in mind to avoid a standard interpretation of ip\_len, which includes header, options, and data. ip\_len is also changed because the code is skipped because this is not a fragment.

- 305-307

ipintr copies the MF flag into the low-order bit of ip\_tos and overlays ip\_tos (&= ~1 clears the low-order bit). ip\_tos must be cast to a pointer to an ipasfrag structure, which contains a valid member. [Section 10.6](#) and [Figure 10.14](#) show the structure.

Although RFC 1122 requires the IP layer to set ip\_tos to 0, that enables the transport layer to set ip\_tos to 1 in a datagram, it only recommends that the IP layer set ip\_tos values to the transport layer at the destination. The low-order bit of the TOS field must always be 0.

hold the MF bit while ip\_off (where the MF is used by the reassembly algorithm).

ip\_off can now be accessed as a 16-bit offset and a 13-bit offset.

- 308

ip\_off is multiplied by 8 to convert from 8-bytes to bytes.

ipf\_mff and ip\_off determine if ipintr should attempt reassembly. Figure 10.12 describes the different cases and actions. Remember that fp points to the list of fragments that has previously received for the datagram. Most of the logic is in ip\_reass.

**Figure 10.12. IP fragment processing in ipintr**

ip_off	ipf_mff	fp	Description	Action
0	false	null	complete datagram	no assembly required
0	false	nonnull	complete datagram	discard the previous fragments
any	true	null	fragment of new datagram	initialize new fragment list with this fragment
any	true	nonnull	fragment of incomplete datagram	insert into existing fragment list, attempt reassembly
nonzero	false	null	tail fragment of new datagram	initialize new fragment list
nonzero	false	nonnull	tail fragment of incomplete datagram	insert into existing fragment list, attempt reassembly

309-322

If ip\_reass is able to assemble a complete data structure, it will update the current fragment with previously received fragments.

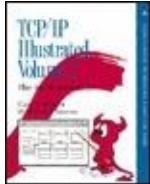
pointer to the reassembled datagram. If reassigned, ip\_reass saves the fragment and ipintr jumps to the next packet ([Figure 8.12](#)).

323-324

This else branch is taken when a complete datagram is received and ip\_hlen is modified as described earlier. This is most received datagrams are not fragments.

If a complete datagram is available after reassembly, it is passed up to the appropriate transport protocol.

```
(*inetsw[ip_protos[ip->ip_p]].pr
```



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.6 ip\_reass Function

ipintr passes ip\_reass a fragment to be processed, and a pointer to the matching reassembly header from ipq. ip\_reass attempts to assemble and return a complete datagram or links the fragment into the datagram's reassembly list for reassembly when the remaining fragments arrive. The head of each reassembly list is an ipq structure, show in Figure 10.13.

**Figure 10.13. ipq structure.**

```
52 struct ipq {  
53     struct ipq *next, *prev;      /* to other reass headers */  
54     u_char    ipq_ttl;          /* time for reass q to live */  
55     u_char    ipq_p;            /* protocol of this fragment */  
56     u_short   ipq_id;          /* sequence id for reassembly */  
57     struct ipasfrag *ipq_next, *ipq_prev;  
58     /* to ip headers of fragments */  
59     struct in_addr ipq_src, ipq_dst;  
60 };
```

— ip\_var.h

## 52-60

The four fields required to identify a datagram's fragments, ip\_id, ip\_p, ip\_src, and ip\_dst, are kept in the ipq structure at the head of each reassembly list. Net/3 constructs the list of datagrams with next and prev and the list of fragments with ipq\_next and ipq\_prev.

The IP header of incoming IP packets is converted to an ipasfrag structure ([Figure 10.14](#)) before it is placed on a reassembly list.

**Figure 10.14. ipasfrag structure.**

```

66 struct ipasfrag {
67 #if BYTE_ORDER == LITTLE_ENDIAN
68     u_char ip_hl:4,
69         ip_v:4;
70 #endif
71 #if BYTE_ORDER == BIG_ENDIAN
72     u_char ip_v:4,
73         ip_hl:4;
74 #endif
75     u_char ipf_mff;      /* XXX overlays ip_tos: use low bit
76                         * to avoid destroying tos;
77                         * copied from (ip_off&IP_MP) */
78     short ip_len;
79     u_short ip_id;
80     short ip_off;
81     u_char ip_ttl;
82     u_char ip_p;
83     u_short ip_sum;
84     struct ipasfrag *ipf_next; /* next fragment */
85     struct ipasfrag *ipf_prev; /* previous fragment */
86 };

```

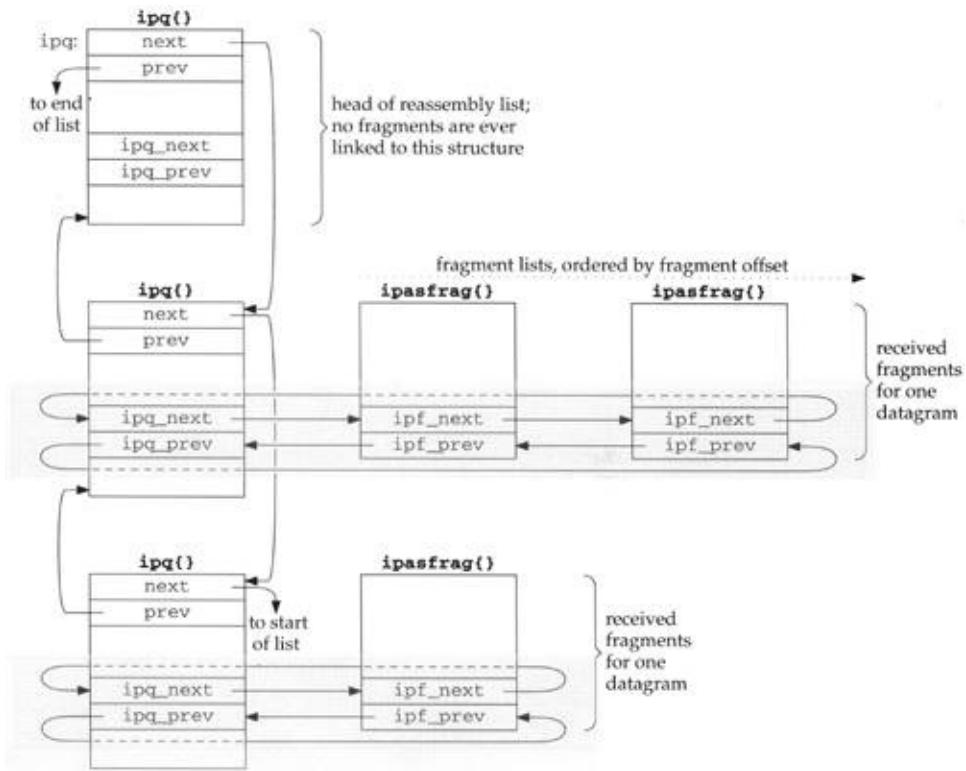
*ip\_var.h*

## 66-86

ip\_reass collects fragments for a particular datagram on a circular doubly linked list joined by the ipf\_next and ipf\_prev members. These pointers overlay the source and destination addresses in the IP header. The ipf\_mff member overlays ip\_tos from the ip structure. The other members are the same.

[Figure 10.15](#) illustrates the relationship between the fragment header list (ipq) and the fragments (ipasfrag).

**Figure 10.15. The fragment header list, ipq, and fragments.**



Down the left side of Figure 10.15 is the list of reassembly headers. The first node in the list is the global ipq structure, ipq. It never has a fragment list associated with it. The ipq list is a doubly linked list used to support fast insertions and deletions. The next and prev pointers reference the next or previous ipq structure, which we have shown by terminating the arrows at the corners of the structures.

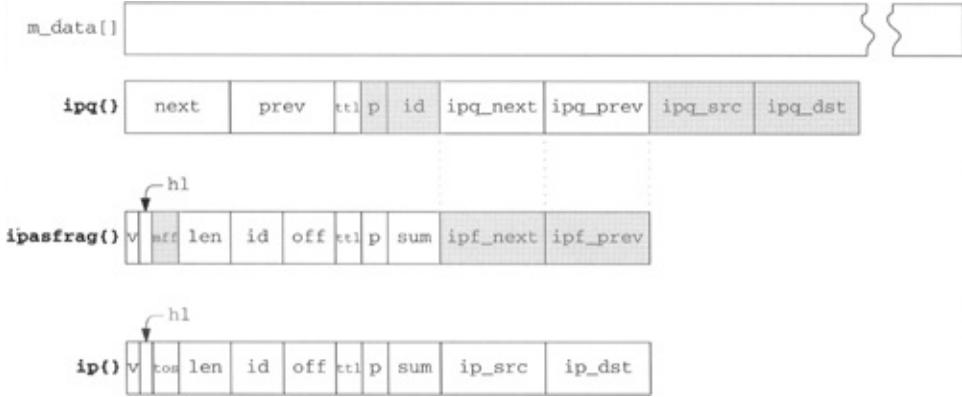
Each ipq structure is the head node of a circular doubly linked list of ipasfrag structures. Incoming fragments are placed

on these fragment lists ordered by their fragment offset. We've highlighted the pointers for these lists in [Figure 10.15](#).

[Figure 10.15](#) still does not show all the complexity of the reassembly structures. The reassembly code is difficult to follow because it relies so heavily on casting pointers to three different structures on the underlying mbuf. We've seen this technique already, for example, when an ip structure overlays the data portion of an mbuf.

[Figure 10.16](#) illustrates the relationship between an mbuf, an ipq structure, an ipasfrag structure, and an ip structure.

**Figure 10.16. An area of memory can be accessed through multiple structures.**



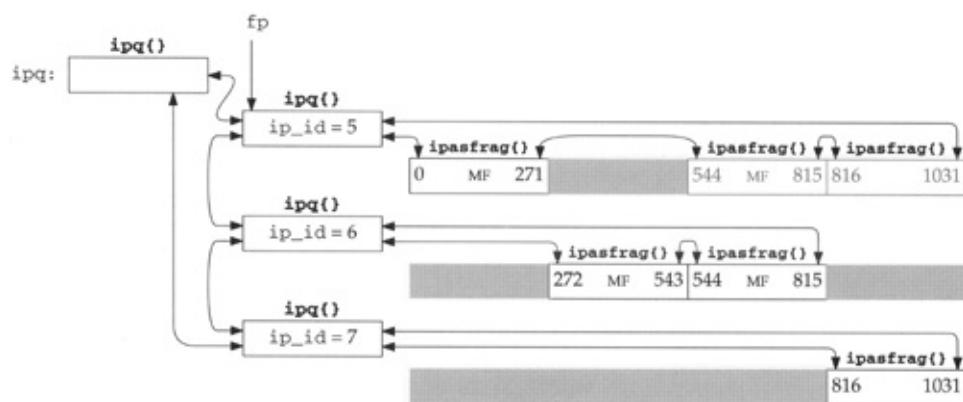
A lot of information is contained within Figure 10.16:

- All the structures are located within the data area of an mbuf.
- The ipq list consists of ipq structures joined by next and prev. Within the structure, the four fields that uniquely identify an IP datagram are saved (shaded in Figure 10.16).
- Each ipq structure is treated as an ipasfrag structure when accessed as the head of a linked list of fragments. The fragments are joined by ipf\_next and ipf\_prev, which overlay the ipq structures' ipq\_next and ipq\_prev members.
- Each ipasfrag structure overlays the ip

structure from the incoming fragment. The data that arrived with the fragment follows the structure in the mbuf. The members that have a different meaning in the ipasfrag structure than they do in the ip structure are shaded.

[Figure 10.15](#) showed the physical connections between the reassembly structures and [Figure 10.16](#) illustrated the overlay technique used by ip\_reass. In [Figure 10.17](#) we show the reassembly structures from a logical point of view: this figure shows the reassembly of three datagrams and the relationship between the ipq list and the ipasfrag structures.

## Figure 10.17. Reassembly of three IP datagrams.



The head of each reassembly list contains the id, protocol, source, and destination address of the original datagram. Only the ip\_id field is shown in the figure. Each fragment list is ordered by the offset field, the fragment is labeled with MF if the MF bit is set, and missing fragments appear as shaded boxes. The numbers within each fragment show the starting and ending byte offset for the fragment relative to the *data portion* of the original datagram, not to the IP header of the original datagram.

The example is constructed to show three UDP datagrams with no IP options and 1024 bytes of data each. The total length of each datagram is 1052 (20 + 8 + 1024) bytes, which is well within the 1500-byte MTU of an Ethernet. The datagrams encounter a SLIP link on the way to the destination, and the router at that link fragments the datagrams to fit within a typical 296-byte SLIP MTU. Each datagram arrives as four fragments. The first fragment contain a standard 20-byte IP header, the 8-byte UDP header, and 264 bytes of data. The second and third fragments contain a 20-byte IP header and

272 bytes of data. The last fragment has a 20-byte header and 216 bytes of data ( $1032 = 272 \times 3 + 216$ ).

In [Figure 10.17](#), datagram 5 is missing a single fragment containing bytes 272 through 543. Datagram 6 is missing the first fragment, bytes 0 through 271, and the end of the datagram starting at offset 816. Datagram 7 is missing the first three fragments, bytes 0 through 815.

[Figure 10.18](#) lists ip\_reass. Remember that ipintr calls ip\_reass when an IP fragment has arrived for this host, and after any options have been processed.

### **Figure 10.18. ip\_reass function: datagram reassembly.**

---

```

337 /*
338 * Take incoming datagram fragment and try to
339 * reassemble it into whole datagram. If a chain for
340 * reassembly of this datagram already exists, then it
341 * is given as fp; otherwise have to make a chain.
342 */
343 struct ip *
344 ip_reass(ip, fp)
345 struct ipasfrag *ip;
346 struct ipq *fp;
347 {
348     struct mbuf *m = dtom(ip);
349     struct ipasfrag *q;
350     struct mbuf *t;
351     int     hlen = ip->ip_hl << 2;
352     int     i, next;
353     /*
354     * Presence of header sizes in mbufs
355     * would confuse code below.
356     */
357     m->m_data += hlen;
358     m->m_len -= hlen;

                                /* reassembly code */

465 dropfrag:
466     ipstat.ips_fragdropped++;
467     m_freem(m);
468     return (0);
469 }

```

---

*ip\_input.c*

## 343-358

When `ip_reass` is called, `ip` points to the fragment and `fp` either points to the matching `ipq` structure or is null.

Since reassembly involves only the data portion of each fragment, `ip_reass` adjusts `m_data` and `m_len` from the `mbuf` containing the fragment to exclude the IP header in each fragment.

## 465-469

When an error occurs during reassembly,

the function jumps to dropfrag, which increments ips\_fragdropped, discards the fragment, and returns a null pointer.

Dropping fragments usually incurs a serious performance penalty at the transport layer since the entire datagram must be retransmitted. TCP is careful to avoid fragmentation, but a UDP application must take steps to avoid fragmentation on its own. [Kent and Mogul 1987] explain why fragmentation should be avoided.

All IP implementations must be able to reassemble a datagram of up to 576 bytes. There is no general way to determine the size of the largest datagram that can be reassembled by a remote host. We'll see in Section 27.5 that TCP has a mechanism to determine the size of the maximum datagram that can be processed by the remote host. UDP has no such mechanism, so many UDP-based protocols (e.g., RIP, TFTP, BOOTP, SNMP, and DNS) are designed around the 576-byte limit.

We'll show the reassembly code in seven parts, starting with Figure 10.19.

## Figure 10.19. ip\_reass function: create reassembly list.

```
359  /*
360   * If first fragment to arrive, create a reassembly queue.
361   */
362  if (fp == 0) {
363      if ((t = m_get(M_DONTWAIT, MT_FTABLE)) == NULL)
364          goto dropfrag;
365      fp = mtod(t, struct ipq *);
366      insque(fp, &ipq);
367      fp->ipq_ttl = IPFRAGTTL;
368      fp->ipq_p = ip->ip_p;
369      fp->ipq_id = ip->ip_id;
370      fp->ipq_next = fp->ipq_prev = (struct ipasfrag *) fp;
371      fp->ipq_src = ((struct ip *) ip)->ip_src;
372      fp->ipq_dst = ((struct ip *) ip)->ip_dst;
373      q = (struct ipasfrag *) fp;
374      goto insert;
375 }
```

*ip\_input.c*

*ip\_input.c*

## Create reassembly list

359-366

When fp is null, ip\_reass creates a reassembly list with the first fragment of the new datagram. It allocates an mbuf to hold the head of the new list (an ipq structure), and calls insque to insert the structure in the list of reassembly lists.

Figure 10.20 lists the functions that manipulate the datagram and fragment lists.

## Figure 10.20. Queueing functions used by ip\_reass.

Function	Description
insque	Insert <i>node</i> just after <i>prev</i> . <code>void insque(void *node, void *prev);</code>
remque	Remove <i>node</i> from list. <code>void remque(void *node);</code>
ip_eng	Insert fragment <i>p</i> just after fragment <i>prev</i> . <code>void ip_eng(struct ipasfrag *p, struct ipasfrag *prev);</code>
ip_deq	Remove fragment <i>p</i> . <code>void ip_deq(struct ipasfrag *p);</code>

The functions insque and remque are defined in machdep.c for the 386 version of Net/3. Each machine has its own machdep.c file in which customized versions of kernel functions are defined, typically to improve performance. This file also contains architecture-dependent functions such as the interrupt handler support, cpu and device configuration, and memory management functions.

insque and remque exist primarily to maintain the kernel's run queue. Net/3 can use them for the datagram reassembly list because both lists have

next and previous pointers as the first two members of their respective node structures. These functions work for any similarly structured list, although the compiler may issue some warnings. This is yet another example of accessing memory through two different structures.

In all the kernel structures the next pointer always precedes the previous pointer ([Figure 10.14](#), for example). This is because the insque and remque functions were first implemented on the VAX using the insque and remque hardware instructions, which require this ordering of the forward and backward pointers.

The fragment lists are not joined with the first two members of the ipasfrag structures ([Figure 10.14](#)) so Net/3 calls ip\_enq and ip\_deq instead of insque and remque.

## Reassembly timeout

The time-to-live field (`ipq_ttl`) is required by RFC 1122 and limits the time Net/3 waits for fragments to complete a datagram. It is different from the TTL field in the IP header, which limits the amount of time a packet circulates in the internet. The IP header TTL field is reused as the reassembly timeout since the header TTL is not needed once the fragment arrives at its final destination.

In Net/3, the initial value of the reassembly timeout is 60 (IPFRAGTTL). Since `ipq_ttl` is decremented every time the kernel calls `ip_slowtimo` and the kernel calls `ip_slowtimo` every 500 ms, the system discards an IP reassembly list if it hasn't assembled a complete IP datagram within 30 seconds of receiving any one of the datagram's fragments. The reassembly timer starts ticking on the first call to `ip_slowtimo` after the list is created.

RFC 1122 recommends that the reassembly time be between 60 and 120 seconds and that an ICMP time exceeded

error be sent to the source host if the timer expires and the first fragment of the datagram has been received. The header and options of the other fragments are always discarded during reassembly and an ICMP error must contain the first 64 bits of the erroneous datagram (or less if the datagram was shorter than 8 bytes). So, if the kernel hasn't received fragment 0, it can't send an ICMP message.

Net/3's timer is a bit too short and Net/3 neglects to send the ICMP message when a fragment is discarded. The requirement to return the first 64 bits of the datagram ensures that the first portion of the transport header is included, which allows the error message to be returned to the application that generated it. Note that TCP and UDP purposely put their port numbers in the first 8 bytes of their headers for this reason.

## Datagram identifiers

368-375

ip\_reass saves ip\_p, ip\_id, ip\_src, and ip\_dst in the ipq structure allocated for this datagram, points the ipq\_next and ipq\_prev pointers to the ipq structure (i.e., it constructs a circular list with one node), points q at this structure, and jumps to insert ([Figure 10.25](#)) where it inserts the first fragment, ip, into the new reassembly list.

The next part of ip\_reass, shown in [Figure 10.21](#), is executed when fp is not null and locates the correct position in the existing list for the new fragment.

### Figure 10.21. ip\_reass function: find position in reassembly list.

```
376  /*                                         ip_input.c
377  * Find a fragment which begins after this one does.
378  */
379  for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next)
380      if (q->ip_off > ip->ip_off)
381          break;
```

376-381

Since fp is not null, the for loop searches the datagram's fragment list to locate a fragment with an offset greater than

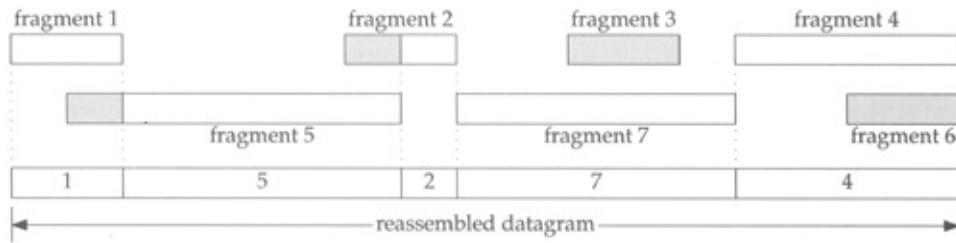
`ip_off`.

The byte ranges contained within fragments may overlap at the destination. This can happen when a transport-layer protocol retransmits a datagram that gets sent along a route different from the one followed by the original datagram. The fragmentation pattern may also be different resulting in overlaps at the destination. The transport protocol must be able to force IP to use the original ID field in order for the datagram to be recognized as a retransmission at the destination.

Net/3 does not provide a mechanism for a transport protocol to ensure that IP ID fields are reused on a retransmitted datagram. `ip_output` always assigns a new value by incrementing the global integer `ip_id` when preparing a new datagram ([Figure 8.22](#)). Nevertheless, a Net/3 system could receive overlapping fragments from a system that lets the transport layer retransmit IP datagrams with the same ID field.

[Figure 10.22](#) illustrates the different ways in which the fragment may overlap with existing fragments. The fragments are numbered according to the order in which they *arrive* at the destination host. The reassembled fragment is shown at the bottom of [Figure 10.22](#). The shaded areas of the fragments are the duplicate bytes that are discarded.

### **Figure 10.22. The byte range of fragments may overlap at the destination.**



In the following discussion, an *earlier* fragment is a fragment that previously arrived at the host.

The code in [Figure 10.23](#) trims or discards incoming fragments.

## Figure 10.23. ip\_reass function: trim incoming packet.

```
382  /* ip_input.c
383  * If there is a preceding fragment, it may provide some of
384  * our data already. If so, drop the data from the incoming
385  * fragment. If it provides all of our data, drop us.
386  */
387  if (q->ipf_prev != (struct ipasfrag *) fp) {
388      i = q->ipf_prev->ip_off + q->ipf_prev->ip_len - ip->ip_off;
389      if (i > 0) {
390          if (i >= ip->ip_len)
391              goto dropfrag;
392          m_adj(dtom(ip), i);
393          ip->ip_off += i;
394          ip->ip_len -= i;
395      }
396 }
```

382-396

ip\_reass discards bytes that overlap the end of an earlier fragment by trimming the new fragment (the front of fragment 5 in [Figure 10.22](#)) or discarding the new fragment (fragment 6) if all its bytes arrived in an earlier fragment (fragment 4).

The code in [Figure 10.24](#) trims or discards existing fragments.

## Figure 10.24. ip\_reass function: trim existing packets.

---

```

397  /*
398   * While we overlap succeeding fragments trim them or,
399   * if they are completely covered, dequeue them.
400   */
401  while (q != (struct ipasfrag *) fp && ip->ip_off + ip->ip_len > q->ip_off) {
402      i = (ip->ip_off + ip->ip_len) - q->ip_off;
403      if (i < q->ip_len) {
404          q->ip_len -= i;
405          q->ip_off += i;
406          m_adj(dtom(q), i);
407          break;
408      }
409      q = q->ipf_next;
410      m_freem(dtom(q->ipf_prev));
411      ip_deq(q->ipf_prev);
412  }

```

---

*ip\_input.c*

**397-412**

If the current fragment partially overlaps the front of an earlier fragment, the duplicate data is trimmed from the earlier fragment (the front of fragment 2 in [Figure 10.22](#)). Any earlier fragments that are completely overlapped by the arriving fragment are discarded (fragment 3).

In [Figure 10.25](#), the incoming fragment is inserted into the reassembly list.

**Figure 10.25. ip\_reass function: insert packet.**

```
413     insert:  
414     /*  
415      * Stick new fragment in its place;  
416      * check for complete reassembly.  
417      */  
418     ip_enq(ip, q->ipf_prev);  
419     next = 0;  
420     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = q->ipf_next) {  
421         if (q->ip_off != next)  
422             return (0);  
423         next += q->ip_len;  
424     }  
425     if (q->ipf_prev->ipf_mff & 1)  
426         return (0);  
----- ip_input.c
```

## 413-426

After trimming, ip\_enq inserts the fragment into the list and the list is scanned to determine if all the fragments have arrived. If any fragment is missing, or the last fragment in the list has ipf\_mff set, ip\_reass returns 0 and waits for more fragments.

When the current fragment completes a datagram, the entire list is converted to an mbuf chain by the code shown in [Figure 10.26](#).

**Figure 10.26. ip\_reass function:  
reassemble datagram.**

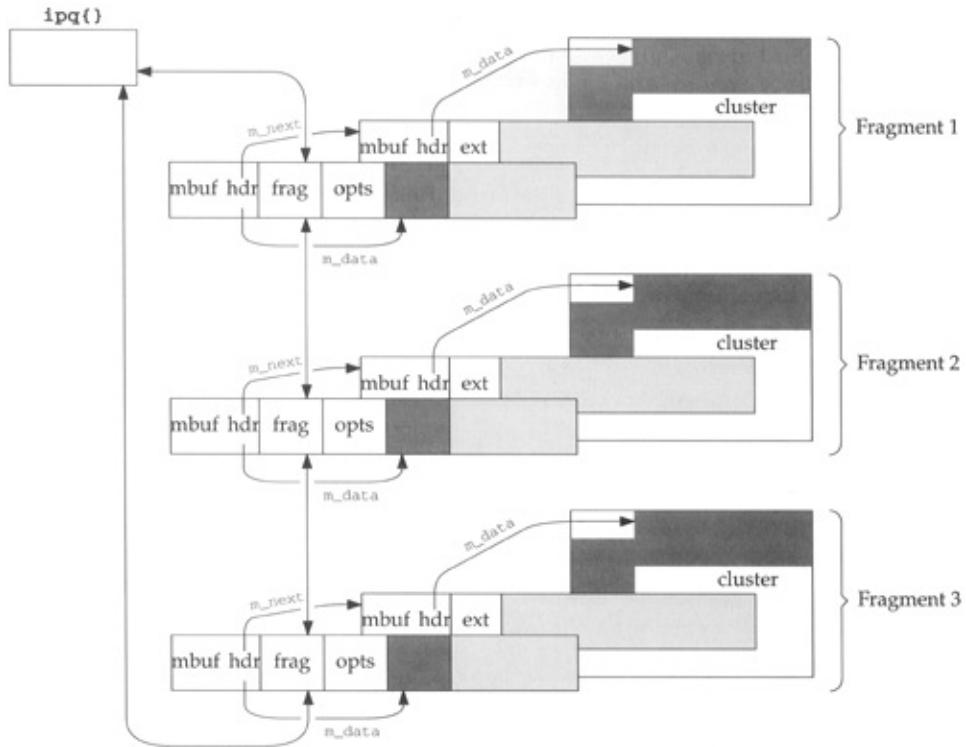
```
427  /*  
428   * Reassembly is complete; concatenate fragments.  
429   */  
430   q = fp->ipq_next;  
431   m = dtom(q);  
432   t = m->m_next;  
433   m->m_next = 0;  
434   m_cat(m, t);  
435   q = q->ipf_next;  
436   while (q != (struct ipasfrag *) fp) {  
437     t = dtom(q);  
438     q = q->ipf_next;  
439     m_cat(m, t);  
440   }  
----- ip_input.c
```

## 427-440

If all the fragments for the datagram have been received, the while loop reconstructs the datagram from the fragments with `m_cat`.

[Figure 10.27](#) shows the relationships between mbufs and the ipq structure for a datagram composed of three fragments.

**Figure 10.27. `m_cat` reassembles the fragments within mbufs.**



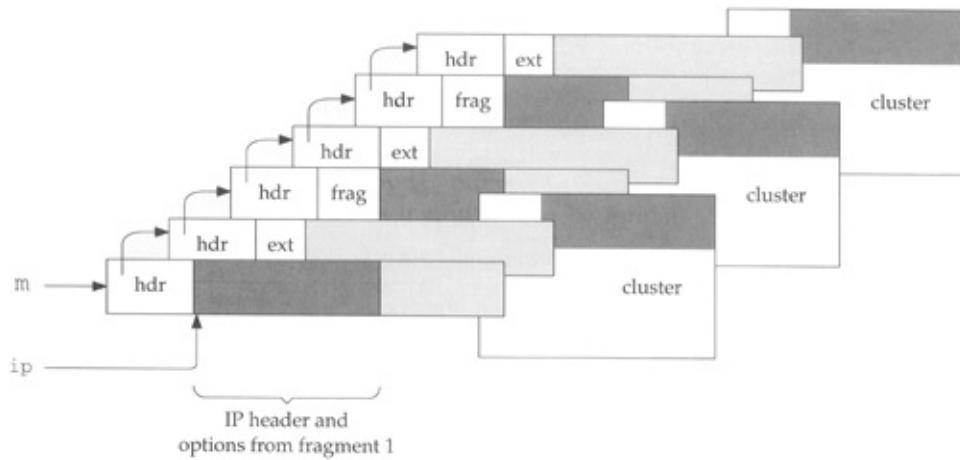
The darkest areas in the figure mark the data portions of a packet and the lighter shaded areas mark the unused portions of the mbufs. We show three fragments each contained in a chain of two mbufs; a packet header, and a cluster. The `m_data` pointer in the first mbuf of each fragment points to the packet data, not the packet header. Therefore, the mbuf chain constructed by `m_cat` includes only the data portion of the fragments.

This is the typical scenario when a fragment contains more than 208 bytes of data ([Section 2.6](#)). The "frag" portion of

the mbufs is the IP header from the fragment. The m\_data pointer of the first mbuf in each chain points beyond "opts" because of the code in [Figure 10.18](#).

[Figure 10.28](#) shows the reassembled datagram using the mbufs from all the fragments. Notice that the IP header and options from fragments 2 and 3 are not included in the reassembled datagram.

## [Figure 10.28. The reassembled datagram.](#)



The header of the first fragment is still being used as an ipasfrag structure. It is restored to a valid IP datagram header by the code shown in [Figure 10.29](#).

## Figure 10.29. ip\_reass function: datagram reassembly.

```
441  /*  
442   * Create header for new ip packet by  
443   * modifying header of first packet;  
444   * dequeue and discard fragment reassembly header.  
445   * Make header visible.  
446   */  
447   ip = fp->ipq_next;  
448   ip->ip_len = next;  
449   ip->ipf_mff &= ~1;  
450   ((struct ip *) ip)->ip_src = fp->ipq_src;  
451   ((struct ip *) ip)->ip_dst = fp->ipq_dst;  
452   remque(fp);  
453   (void) m_free(dtom(fp));  
454   m = dtom(ip);  
455   m->m_len += (ip->ip_hl << 2);  
456   m->m_data -= (ip->ip_hl << 2);  
457   /* some debugging cruft by sklower, below, will go away soon */  
458   if (m->m_flags & M_PKTHDR) { /* XXX this should be done elsewhere */  
459     int plen = 0;  
460     for (t = m; m = m->m_next)  
461       plen += m->m_len;  
462     t->m_pkthdr.len = plen;  
463   }  
464   return ((struct ip *) ip);
```

## Reconstruct datagram header

441-456

ip\_reass points ip to the first fragment in the list and changes the ipasfrag structure back to an ip structure by restoring the length of the datagram to ip\_len, the source address to ip\_src, the destination address to ip\_dst; and by clearing the low-order bit in ipf\_mff. (Recall from [Figure 10.14](#) that ipf\_mff in the ipasfrag structure

overlays ipf\_tos in the ip structure.)

ip\_reass removes the entire packet from the reassembly list with remque, discards the ipq structure that was the head of the list, and adjusts m\_len and m\_data in the first mbuf to include the previously hidden IP header and options from the first fragment.

## Compute packet length

457-464

The code here is always executed, since the first mbuf for the datagram is always a packet header. The for loop computes the number of data bytes in the mbuf chain and saves the value in m\_pkthdr.len.

The purpose of the *copied* bit in the option type field should be clear now. Since the only options retained at the destination are those that appear in the first fragment, only options that control processing of the packet as it travels toward its destination are copied. Options that collect information

while in transit are not copied, since the information collected is discarded at the destination when the packet is reassembled.

---

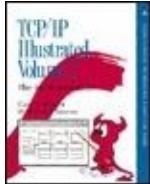
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.7 ip\_slowtimo Function

As shown in [Section 7.4](#), each protocol in Net/3 may specify a function to be called every 500 ms. For IP, that function is `ip_slowtimo`, shown in [Figure 10.30](#), which times out the fragments on the reassembly list.

**Figure 10.30. ip\_slowtimo function.**

```
515 void  
516 ip_slowtimo(void)  
517 {  
518     struct ipq *fp;  
519     int    s = splnet();  
520     fp = ipq.next;  
521     if (fp == 0) {  
522         splx(s);  
523         return;  
524     }  
525     while (fp != &ipq) {  
526         --fp->ipq_ttl;  
527         fp = fp->next;  
528         if (fp->prev->ipq_ttl == 0) {  
529             ipstat.ips_fragtimeout++;  
530             ip_freef(fp->prev);  
531         }  
532     }  
533     splx(s);  
534 }
```

ip\_input.c

## 515-534

ip\_slowtimo traverses the list of partial datagrams and decrements the reassembly TTL field. ip\_freef is called if the field drops to 0 to discard the fragments associated with the datagram. ip\_slowtimo runs at splnet to prevent the lists from being modified by incoming packets.

ip\_freef is shown in [Figure 10.31](#).

**Figure 10.31. ip\_freef function.**

```
474 void ip_freef(fp)
475 struct ipq *fp;
476 {
477     struct ipasfrag *q, *p;
478     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = p) {
479         p = q->ipf_next;
480         ip_deq(q);
481         m_freem(dtom(q));
482     }
483     remque(fp);
484     (void) m_free(dtom(fp));
485 }
486 }
```

ip\_input.c

**470-486**

ip\_freef removes and releases every fragment on the list pointed to by fp and then releases the list itself.

## ip\_drain Function

In [Figure 7.14](#) we showed that IP defines ip\_drain as the function to be called when the kernel needs additional memory. This usually occurs during mbuf allocation, which we described with [Figure 2.13](#). ip\_drain is shown in [Figure 10.32](#).

**Figure 10.32. ip\_drain function.**

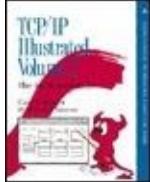
```
538 void  
539 ip_drain()  
540 {  
541     while (ipq.next != &ipq) {  
542         ipstat.ips_fragdropped++;  
543         ip_freef(ipq.next);  
544     }  
545 }
```

---

ip\_input.c

## 538-545

The simplest way for IP to release memory is to discard all the IP fragments on the reassembly list. For IP fragments that belong to a TCP segment, TCP eventually retransmits the data. IP fragments that belong to a UDP datagram are lost and UDP-based protocols must handle this at the application layer.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 10. IP Fragmentation and Reassembly

### 10.8 Summary

In this chapter we showed how `ip_output` splits an outgoing datagram into fragments if it is too large to be transmitted on the selected network. Since fragments may themselves be fragmented as they travel toward their final destination and may take multiple paths, only the destination host can reassemble the original datagram.

`ip_reass` accepts incoming fragments and attempts to reassemble datagrams. If it is successful, the datagram is passed back to `ipintr` and then to the appropriate transport protocol. Every IP implementation must reassemble

datagrams of up to 576 bytes. The only limit for Net/3 is the number of mbufs that are available. `ip_slowtimo` discards incomplete datagrams when all their fragments haven't been received within a reasonable amount of time.

## Exercises

**10.1** Modify `ip_slowtimo` to send an ICMP time exceeded message when it discards an incomplete datagram ([Figure 11.1](#)).

**10.2** The recorded route in a fragmented datagram may be different in each fragment. When a datagram is reassembled at the destination host, which return route is available to the transport protocols?

**10.3** Draw a picture showing the mbufs involved in the ipq structure and its associated fragment list for the

fragment with an ID of 7 in [Figure 10.17](#).

[[Auerbach 1994](#)] suggests that after fragmenting a datagram, the last fragment should be sent first. If the receiving system gets that last fragment first, it can use the offset to allocate an appropriately sized reassembly buffer for the datagram.

**10.4** Modify ip\_output to send the last fragment first.

[[Auerbach 1994](#)] notes that some commercial TCP/IP implementations have been known to crash if they receive the last fragment first.

Use the statistics in [Figure 8.5](#) to answer the following questions. What is the average number of fragments per reassembled datagram? What is the average number of fragments created when an outgoing datagram is fragmented?

**10.6** What happens to a packet when the reserved bit in ip\_off is set?

---

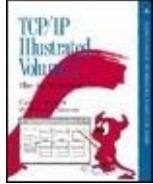
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 11. ICMP: Internet Control Message Protocol

Section 11.1. Introduction

Section 11.2. Code Introduction

Section 11.3. icmp Structure

Section 11.4. ICMP protosw Structure

Section 11.5. Input Processing:  
icmp\_input Function

Section 11.6. Error Processing

Section 11.7. Request Processing

Section 11.8. Redirect Processing

Section 11.9. Reply Processing

Section 11.10. Output Processing

Section 11.11. icmp\_error Function

Section 11.12. icmp\_reflect Function

[Section 11.13. icmp\\_send Function](#)

[Section 11.14. icmp\\_sysctl Function](#)

[Section 11.15. Summary](#)

---

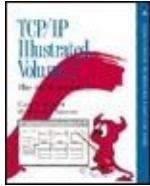
**Team-Fly**



[Previous](#)

[Next](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.1 Introduction

ICMP communicates error and administrative messages between IP systems and is an integral and required part of any IP implementation. The specification for ICMP appears in RFC 792 [[Postel 1981b](#)]. RFC 950 [[Mogul and Postel 1985](#)] and RFC 1256 [[Deering 1991a](#)] define additional ICMP message types. RFC 1122 [[Braden 1989a](#)] also provides important details on ICMP.

ICMP has its own transport protocol number (1) allowing ICMP messages to be carried within an IP datagram. Application programs can send and receive ICMP

messages directly through the raw IP interface discussed in [Chapter 32](#).

We can divide the ICMP messages into two classes: errors and queries. Query messages are defined in pairs: a request and its reply. ICMP error messages always include the IP header (and options) along with at least the first 8 bytes of the data from the initial fragment of the IP datagram that caused the error. The standard assumes that the 8 bytes includes any demultiplexing information from the transport protocol header of the original packet, which allows a transport protocol to deliver an ICMP error to the correct process.

TCP and UDP port numbers appear within the first 8 bytes of their respective headers.

[Figure 11.1](#) shows all the currently defined ICMP messages. The messages above the double line are ICMP requests and replies; those below the double line are ICMP errors.

## Figure 11.1. ICMP message types and codes.

type and code	Description	PRC_
<i>ICMP_ECHO</i> <i>ICMP_ECHOREPLY</i>	echo request echo reply	
<i>ICMP_TSTAMP</i> <i>ICMP_TSTAMPREPLY</i>	timestamp request timestamp reply	
<i>ICMP_MASKREQ</i> <i>ICMP_MASKREPLY</i>	address mask request address mask reply	
<i>ICMP_IREQ</i> <i>ICMP_IREQREPLY</i>	information request (obsolete) information reply (obsolete)	
<i>ICMP_ROUTERADVERT</i> <i>ICMP_ROUTERSOLICIT</i>	router advertisement router solicitation	
<i>ICMP_REDIRECT</i> <i>ICMP_REDIRECT_NET</i> <i>ICMP_REDIRECT_HOST</i> <i>ICMP_REDIRECT_TOSNET</i> <i>ICMP_REDIRECT_TOSHOST</i> other	better route available better route available for network better route available for host better route available for TOS and network better route available for TOS and host unrecognized code	PRC_REDIRECT_HOST PRC_REDIRECT_HOST PRC_REDIRECT_HOST PRC_REDIRECT_HOST PRC_REDIRECT_HOST
<i>ICMP_UNREACH</i> <i>ICMP_UNREACH_NET</i> <i>ICMP_UNREACH_HOST</i> <i>ICMP_UNREACH_PROTOCOL</i> <i>ICMP_UNREACH_PORT</i> <i>ICMP_UNREACH_SRCFAIL</i> <i>ICMP_UNREACH_NEEDFRAG</i> <i>ICMP_UNREACH_NET_UNKNOWN</i> <i>ICMP_UNREACH_HOST_UNKNOWN</i> <i>ICMP_UNREACH_ISOLATED</i> <i>ICMP_UNREACH_NET_PROHIB</i>  <i>ICMP_UNREACH_HOST_PROHIB</i>  <i>ICMP_UNREACH_TOSNET</i> <i>ICMP_UNREACH_TOSHOST</i> 13 14 15 other	destination unreachable network unreachable host unreachable protocol unavailable at destination port inactive at destination source route failed fragmentation needed and DF bit set destination network unknown destination host unknown source host isolated communication with destination network administratively prohibited communication with destination host administratively prohibited network unreachable for type of service host unreachable for type of service communication administratively prohibited by filtering host precedence violation precedence cutoff in effect unrecognized code	PRC_UNREACH_NET PRC_UNREACH_HOST PRC_UNREACH_PROTOCOL PRC_UNREACH_PORT PRC_UNREACH_SRCFAIL PRC_MSGSIZE PRC_UNREACH_NET PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_NET PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_NET PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_NET
<i>ICMP_TIMXCEED</i> <i>ICMP_TIMXCEED_INTRANS</i> <i>ICMP_TIMXCEED_REASS</i> other	time exceeded IP time-to-live expired in transit reassemble time-to-live expired unrecognized code	PRC_TIMXCEED_INTRANS PRC_TIMXCEED_REASS
<i>ICMP_PARAMPROB</i> 0 <i>ICMP_PARAMPROB_OPTABSENT</i> other	problem with IP header unspecified header error required option missing byte offset of invalid byte	PRC_PARAMPROB PRC_PARAMPROB
<i>ICMP_SOURCEQUENCH</i>	request to slow transmission	PRC_QUENCH
other	unrecognized type	

Figures 11.1 and 11.2 contain a lot of information:

## Figure 11.2. ICMP message types and

## codes (continued).

type and code	icmp_input	UDP	TCP	errno
<i>ICMP_ECHO</i>	icmp_reflect			
<i>ICMP_ECHOREPLY</i>	rip_input			
<i>ICMP_TSTAMP</i>	icmp_reflect			
<i>ICMP_TSTAMPREPLY</i>	rip_input			
<i>ICMP_MASKREQ</i>	icmp_reflect			
<i>ICMP_MASKREPLY</i>	rip_input			
<i>ICMP_IREQ</i>	rip_input			
<i>ICMP_IREQREPLY</i>	rip_input			
<i>ICMP_ROUTERADVERT</i>	rip_input			
<i>ICMP_ROUTERSOLICIT</i>	rip_input			
<i>ICMP_REDIRECT</i>				
<i>ICMP_REDIRECT_NET</i>	pfctlinput	in_rtchange	in_rtchange	
<i>ICMP_REDIRECT_HOST</i>	pfctlinput	in_rtchange	in_rtchange	
<i>ICMP_REDIRECT_TOSNET</i>	pfctlinput	in_rtchange	in_rtchange	
<i>ICMP_REDIRECT_TOSHOST</i>	pfctlinput	in_rtchange	in_rtchange	
<i>other</i>	rip_input			
<i>ICMP_UNREACH</i>				
<i>ICMP_UNREACH_NET</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_HOST</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_PROTOCOL</i>	pr_ctlinput	udp_notify	tcp_notify	ECONNREFUSED
<i>ICMP_UNREACH_PORT</i>	pr_ctlinput	udp_notify	tcp_notify	ECONNREFUSED
<i>ICMP_UNREACH_SRCFAIL</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_NEEDFRAG</i>	pr_ctlinput	udp_notify	tcp_notify	EMSGSIZE
<i>ICMP_UNREACH_NET_UNKNOWN</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_HOST_UNKNOWN</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_ISOLATED</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_NET_PROHIB</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_HOST_PROHIB</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_TOSNET</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>ICMP_UNREACH_TOSHOST</i>	pr_ctlinput	udp_notify	tcp_notify	EHOSTUNREACH
<i>13</i>	rip_input			
<i>14</i>	rip_input			
<i>15</i>	rip_input			
<i>other</i>	rip_input			
<i>ICMP_TIMXCEED</i>				
<i>ICMP_TIMXCEED_INTRANS</i>	pr_ctlinput	udp_notify	tcp_notify	
<i>ICMP_TIMXCEED_REASS</i>	pr_ctlinput	udp_notify	tcp_notify	
<i>other</i>	rip_input			
<i>ICMP_PARAMPROB</i>				
<i>0</i>	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
<i>ICMP_PARAMPROB_OPTABSENT</i>	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
<i>other</i>	rip_input			
<i>ICMP_SOURCEQUENCH</i>	pr_ctlinput	udp_notify	tcp_quench	
<i>other</i>	rip_input			

- The PRC\_ column shows the mapping between the ICMP messages and the protocol-independent error codes processed by Net/3 ([Section 11.6](#)). This column is blank for requests and

replies, since no error is generated in that case. If this column is blank for an ICMP error, the code is not recognized by Net/3 and the error message is silently discarded.

- [Figure 11.3](#) shows where we discuss each of the functions listed in [Figure 11.2](#).

### Figure 11.3. Functions called during ICMP input processing.

Function	Description	Reference
icmp_reflect	generate reply to ICMP request	Section 11.12
in_rtchange	update IP routing tables	Figure 22.34
pfctlinput	report error to all protocols	Section 7.7
pr_ctlinput	report error to the protocol associated with the socket	Section 7.4
rip_input	process unrecognized ICMP messages	Section 32.5
tcp_notify	ignore or report error to process	Figure 27.12
tcp_quench	slow down the output	Figure 27.13
udp_notify	report error to process	Figure 23.31

- The icmp\_input column shows the function called by icmp\_input for each ICMP message.
- The UDP column shows the functions that process ICMP messages for UDP sockets.

- The TCP column shows the functions that process ICMP messages for TCP sockets. Note that ICMP source quench errors are handled by `tcp_quench`, not `tcp_notify`.
- If the errno column is blank, the kernel does not report the ICMP message to the process.
- The last line in the tables shows that unrecognized ICMP messages are delivered to the raw IP protocol where they may be received by processes that have arranged to receive ICMP messages.

In Net/3, ICMP is implemented as a transport-layer protocol above IP and does not generate errors or requests; it formats and sends these messages on behalf of the other protocols. ICMP passes incoming errors and replies to the appropriate transport protocol or to processes that are waiting for ICMP messages. On the other hand, ICMP responds to most incoming ICMP requests with an appropriate ICMP reply. [Figure 11.4](#) summarizes this

information.

## Figure 11.4. ICMP message processing.

ICMP message type	Incoming	Outgoing
request	kernel responds with reply	generated by a process
reply	passed to raw IP	generated by kernel
error	passed to transport protocols and raw IP	generated by IP or transport protocols
unknown	passed to raw IP	generated by a process

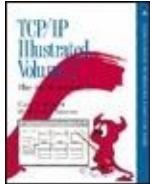
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.2 Code Introduction

The two files listed in [Figure 11.5](#) contain the ICMP data structures, statistics, and processing code described in this chapter.

**Figure 11.5. Files discussed in this chapter.**

File	Description
<code>netinet/ip_icmp.h</code>	ICMP structure definitions
<code>netinet/ip_icmp.c</code>	ICMP processing

### Global Variables

The global variables shown in [Figure 11.6](#)

are introduced in this chapter.

## Figure 11.6. Global variables introduced in this chapter.

Variable	Type	Description
'icmpmaskrepl	int	enables the return of ICMP address mask replies
icmpstat	struct icmpstat	ICMP statistics (Figure 11.7)

## Statistics

Statistics are collected by the members of the icmpstat structure shown in [Figure 11.7](#).

## Figure 11.7. Statistics collected in this chapter.

icmpstat member	Description	Used by SNMP
icps_olddicmp	#errors discarded because datagram was an ICMP message	•
icps_oldshort	#errors discarded because IP datagram was too short	•
icps_badcode	#ICMP messages discarded because of an invalid code	•
icps_badlen	#ICMP messages discarded because of an invalid ICMP body	•
icps_checksum	#ICMP messages discarded because of a bad ICMP checksum	•
icps_tooshort	#ICMP messages discarded because of a short ICMP header	•
icps_outhist[]	array of output counters; one for each ICMP type	•
icps_inhist[]	array of input counters; one for each ICMP type	•
icps_error	#of calls to icmp_error (excluding redirects)	
icps_reflect	#ICMP messages reflected by the kernel	

We'll see where these counters are

incremented as we proceed through the code.

Figure 11.8 shows some sample output of these statistics, from the netstat -s command.

### Figure 11.8. Sample ICMP statistics.

netstat -s output	icmpstat member
84124 calls to icmp_error	icps_error
0 errors not generated 'cuz old message was icmp	icps_olddicmp
Output histogram:	icps_outhist[]
echo reply: 11770	ICMP_ECHOREPLY
destination unreachable: 84118	ICMP_UNREACH
time exceeded: 6	ICMP_TIMXCEED
6 messages with bad code fields	icps_badcode
0 messages < minimum length	icps_badlen
0 bad checksums	icps_checksum
143 messages with bad length	icps_tooshort
Input histogram:	icps_inhist[]
echo reply: 793	ICMP_ECHOREPLY
destination unreachable: 305869	ICMP_UNREACH
source quench: 621	ICMP_SOURCEQUENCH
routing redirect: 103	ICMP_REDIRECT
echo: 11770	ICMP_ECHO
time exceeded: 25296	ICMP_TIMXCEED
11770 message responses generated	icps_reflect

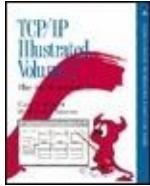
## SNMP Variables

Figure 11.9 shows the relationship between the variables in the SNMP ICMP group and the statistics collected by Net/3.

## Figure 11.9. Simple SNMP variables in ICMP group.

SNMP variable	icmpstat member	Description
icmpInMsgs	see text	#ICMP messages received
icmpInErrors	icps_badcode + icps_badlen + icps_checksum + icps_tooshort	#ICMP messages discarded because of an error
icmpInDestUnreachs icmpInTimeExcds icmpInParmProbs icmpInSrcQuenches icmpInRedirects icmpInEchos icmpInEchoReps icmpInTimestamps icmpInTimestampReps icmpInAddrMasks icmpInAddrMaskReps	icps_inhist[] counter	#ICMP messages received for each type
icmpOutMsgs icmpOutErrors	see text icps_oldicmp + icps_oldshort	#ICMP messages sent #ICMP errors not sent because of an error
icmpOutDestUnreachs icmpOutTimeExcds icmpOutParmProbs icmpOutSrcQuenches icmpOutRedirects icmpOutEchos icmpOutEchoReps icmpOutTimestamps icmpOutTimestampReps icmpOutAddrMasks icmpOutAddrMaskReps	icps_outhist[] counter	#ICMP messages sent for each type

**icmpInMsgs** is the sum of the counts in the **icps\_inhist** array and **icmpInErrors**, and **icmpOutMsgs** is the sum of the counts in the **icps\_outhist** array and **icmpOutErrors**.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.3 icmp Structure

Net/3 accesses an ICMP message through the icmp structure shown in [Figure 11.10](#).

**Figure 11.10. icmp structure.**

---

```

42 struct icmp {
43     u_char    icmp_type;           /* type of message, see below */
44     u_char    icmp_code;          /* type sub code */
45     u_short   icmp_cksum;        /* ones complement cksum of struct */
46     union {
47         u_char   ih_pptr;         /* ICMP_PARAMPROB */
48         struct in_addr ih_gwaddr; /* ICMP_REDIRECT */
49         struct ih_idseq {
50             n_short icd_id;
51             n_short icd_seq;
52         } ih_idseq;
53         int      ih_void;
54     /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
55     struct ih_pmtu {
56         n_short ipm_void;
57         n_short ipm_nextmtu;
58     } ih_pmtu;
59 } icmp_hun;
60 #define icmp_pptr  icmp_hun.ih_pptr
61 #define icmp_gwaddr icmp_hun.ih_gwaddr
62 #define icmp_id    icmp_hun.ih_idseq.icd_id
63 #define icmp_seq   icmp_hun.ih_idseq.icd_seq
64 #define icmp_void  icmp_hun.ih_void
65 #define icmp_pmvoid icmp_hun.ih_pmtu.ipm_void
66 #define icmp_nextmtu  icmp_hun.ih_pmtu.ipm_nextmtu
67     union {
68         struct id_ts {
69             n_time  its_otime;
70             n_time  its_rtime;
71             n_time  its_ttime;
72         } id_ts;
73         struct id_ip {
74             struct ip idi_ip;
75             /* options and then 64 bits of data */
76         } id_ip;
77         u_long   id_mask;
78         char    id_data[1];
79     } icmp_dun;
80 #define icmp_otime  icmp_dun.id_ts.its_otime
81 #define icmp_rtime  icmp_dun.id_ts.its_rtime
82 #define icmp_ttime  icmp_dun.id_ts.its_ttime
83 #define icmp_ip    icmp_dun.id_ip.idi_ip
84 #define icmp_mask  icmp_dun.id_mask
85 #define icmp_data  icmp_dun.id_data
86 };

```

---

ip\_icmp.h

## 42-45

icmp\_type identifies the particular message, and icmp\_code further specifies the message (the first column of [Figure 11.1](#)). icmp\_cksum is computed with the same algorithm as the IP header checksum and protects the entire ICMP

message (not just the header as with IP).

46-79

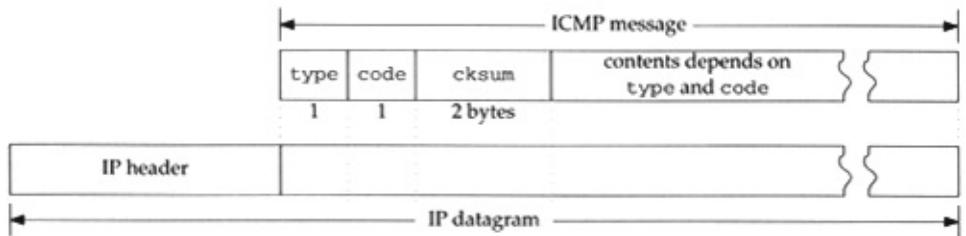
The unions icmp\_hun (header union) and icmp\_dun (data union) access the various ICMP messages according to icmp\_type and icmp\_code. Every ICMP message uses icmp\_hun; only some utilize icmp\_dun. Unused fields must be set to 0.

80-86

As we have seen with other nested structures (e.g., mbuf, le\_softc, and ether\_arp) the #define macros simplify access to structure members.

[Figure 11.11](#) shows the overall structure of an ICMP message and reiterates that an ICMP message is encapsulated within an IP datagram. We show the specific structure of each message when we encounter it in the code.

**Figure 11.11. An ICMP message (icmp omitted).**



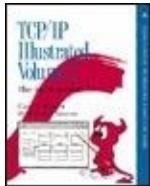
---

**Team-Fly** 

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.4 ICMP protosw Structure

The protosw structure in `inetsw[4]` ([Figure 7.13](#)) describes ICMP and supports both kernel and process access to the protocol. We show this structure in [Figure 11.12](#). Within the kernel, incoming ICMP messages are processed by `icmp_input`. Outgoing ICMP messages generated by processes are handled by `rip_output`. The three functions beginning with `rip_` are described in [Chapter 32](#).

**Figure 11.12. ICMP inetsw entry.**

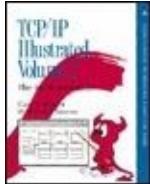
Member	inetsw[4]	Description
pr_type	<i>SOCK_RAW</i>	ICMP provides raw packet services
pr_domain	<i>&amp;inetdomain</i>	ICMP is part of the Internet domain
pr_protocol	<i>IPPROTO_ICMP (1)</i>	appears in the ip_p field of the IP header
pr_flags	<i>PR_ATOMIC PR_ADDR</i>	socket layer flags, not used by ICMP
pr_input	<i>icmp_input</i>	receives ICMP messages from the IP layer
pr_output	<i>rip_output</i>	sends ICMP messages to the IP layer
pr_ctlinput	<i>0</i>	not used by ICMP
pr_ctloutput	<i>rip_ctloutput</i>	respond to administrative requests from a process
pr_usrreq	<i>rip_usrreq</i>	respond to communication requests from a process
pr_init	<i>0</i>	not used by ICMP
pr_fasttimo	<i>0</i>	not used by ICMP
pr_slowtimo	<i>0</i>	not used by ICMP
pr_drain	<i>0</i>	not used by ICMP
pr_sysctl	<i>icmp_sysctl</i>	modify ICMP parameters

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



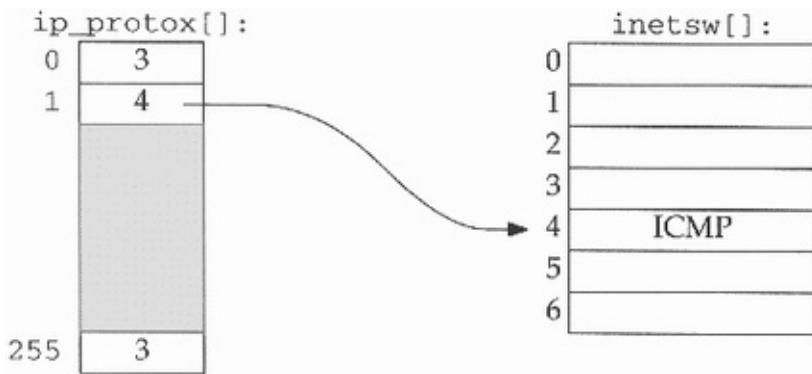
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.5 Input Processing: `icmp_input` Function

Recall that `ipintr` demultiplexes datagrams based on the transport protocol number, `ip_p`, in the IP header. For ICMP messages, `ip_p` is 1, and through `ip_protox`, it selects `inetsw[4]`.

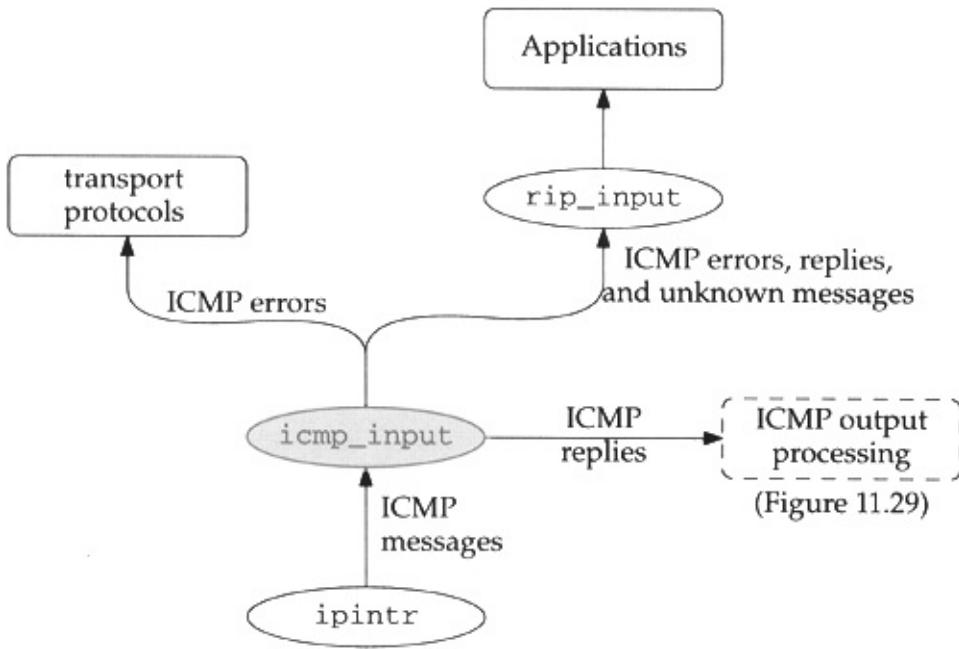
**Figure 11.13. An `ip_p` value of 1 selects `inetsw[4]`.**



The IP layer calls `icmp_input` indirectly through the `pr_input` function of `inetsw[4]` when an ICMP message arrives ([Figure 8.15](#)).

We'll see in `icmp_input` that each ICMP message may be processed up to three times: by `icmp_input`, by the transport protocol associated with the IP packet within an ICMP error message, and by a process that registers interest in receiving ICMP messages. [Figure 11.14](#) shows the overall organization of ICMP input processing.

**Figure 11.14. ICMP input processing.**



We discuss `icmp_input` in five sections: (1) verification of the received message, (2) ICMP error messages, (3) ICMP requests messages, (4) ICMP redirect messages, (5) ICMP reply messages. [Figure 11.15](#) shows the first portion of the `icmp_input` function.

**Figure 11.15. `icmp_input` function.**

---

*ip\_icmp.c*

```
131 static struct sockaddr_in icmpssrc = { sizeof (struct sockaddr_in), AF_INET };
132 static struct sockaddr_in icmpdst = { sizeof (struct sockaddr_in), AF_INET };
133 static struct sockaddr_in icmpgw = { sizeof (struct sockaddr_in), AF_INET };
134 struct sockaddr_in icmpmask = { 8, 0 };

135 void
136 icmp_input(m, hlen)
137 struct mbuf *m;
138 int     hlen;
139 {
140     struct icmp *icp;
141     struct ip *ip = mtod(m, struct ip *);
142     int     icmplen = ip->ip_len;
143     int     i;
144     struct in_ifaddr *ia;
145     void    (*ctlfunc) (int, struct sockaddr *, struct ip *);
146     int     code;
147     extern u_char ip_proto[];

148     /*
149      * Locate icmp structure in mbuf, and check
150      * that not corrupted and of at least minimum length.
151      */
152     if (icmplen < ICMP_MINLEN) {
153         icmpstat.icps_tooshort++;
154         goto freeit;
155     }
156     i = hlen + min(icmplen, ICMP_ADVLENMIN);
157     if (m->m_len < i && (m = m_pullup(m, i)) == 0) {
158         icmpstat.icps_tooshort++;
159         return;
160     }
161     ip = mtod(m, struct ip *);
162     m->m_len -= hlen;
163     m->m_data += hlen;
164     icp = mtod(m, struct icmp *);
165     if (in_cksum(m, icmplen)) {
166         icmpstat.icps_checksum++;
167         goto freeit;
168     }
169     m->m_len += hlen;
170     m->m_data -= hlen;

171     if (icp->icmp_type > ICMP_MAXTYPE)
172         goto raw;
173     icmpstat.icps_inhist[icp->icmp_type]++;
174     code = icp->icmp_code;
175     switch (icp->icmp_type) {

/* ICMP message processing */

317     default:
318         break;
319     }
320 raw:
321     rip_input(m);
322     return;

323 freeit:
324     m_freem(m);
325 }
```

---

*ip\_icmp.c*

## Static structures

131-134

These four structures are statically allocated to avoid the delays of dynamic allocation every time icmp\_input is called and to minimize the size of the stack since icmp\_input is called at interrupt time when the stack size is limited. icmp\_input uses these structures as temporary variables.

The naming of icmpsrt is misleading since icmp\_input uses it as a temporary sockaddr\_in variable and it never contains a source address. In the Net/2 version of icmp\_input, the source address of the message was copied to icmpsrt at the end of the function before the message was delivered to the raw IP mechanism by the raw\_input function. Net/3 calls rip\_input, which expects only a pointer to the packet, instead of raw\_input. Despite this change, icmpsrt retains its name from Net/2.

## Validate message

135-139

icmp\_input expects a pointer to the datagram containing the received ICMP message (m) and the length of the datagram's IP header in bytes (hlen).

[Figure 11.16](#) lists several constants and a macro that simplify the detection of invalid ICMP messages in icmp\_input.

### Figure 11.16. Constants and a macro referenced by ICMP to validate messages.

Constant/Macro	Value	Description
<code>ICMP_MINLEN</code>	8	minimum size of an ICMP message
<code>ICMP_TSLEN</code>	20	size of ICMP timestamp messages
<code>ICMP_MASKLEN</code>	12	size of ICMP address mask messages
<code>ICMP_ADVLENMIN</code>	36	minimum size of an ICMP error (advise) message $(IP + ICMP + BADIP = 20 + 8 + 8 = 36)$
<code>ICMP_ADVLEN(p)</code>	$36 + \text{optsize}$	size of an ICMP error message including <code>optsize</code> bytes of IP options from the invalid packet <code>p</code> .

140-160

icmp\_input pulls the size of the ICMP message from ip\_len and stores it in icmplen. Remember from [Chapter 8](#) that ipintr excludes the length of the header from ip\_len. If the message is too short to be a valid ICMP message, icps\_tooshort is

incremented and the message discarded. If the ICMP header and the IP header are not contiguous in the first mbuf, `m_pullup` ensures that the ICMP header and the IP header of any enclosed IP packet are in a single mbuf.

## Verify checksum

161-170

`icmp_input` hides the IP header in the mbuf and verifies the ICMP checksum with `in_cksum`. If the message is damaged, `icps_checksum` is incremented and the message discarded.

## Verify type

171-175

If the message type (`icmp_type`) is out of the recognized range, `icmp_input` jumps around the switch to raw ([Section 11.9](#)). If it is in the recognized range, `icmp_input` duplicates `icmp_code` and the switch processes the message according to

icmp\_type.

After the processing within the ICMP switch statement, icmp\_input sends ICMP messages to rip\_input where they are distributed to processes that are prepared to receive ICMP messages. The only messages that are not passed to rip\_input are damaged messages (length or checksum errors) and ICMP request messages, which are handled exclusively by the kernel. In both cases, icmp\_input returns immediately, skipping the code at raw.

## Raw ICMP input

317-325

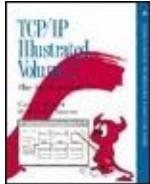
icmp\_input passes the incoming message to rip\_input, which distributes it to listening processes based on the protocol and the source and destination addresses within the message ([Chapter 32](#)).

The raw IP mechanism allows a process to send and to receive ICMP messages

directly, which is desirable for several reasons:

- New ICMP messages can be handled by a process without having to modify the kernel (e.g., router advertisement, [Figure 11.28](#)).
- Utilities for sending ICMP requests and processing the replies can be implemented as a process instead of as a kernel module (ping and traceroute).
- A process can augment the kernel processing of a message. This is common with the ICMP redirect messages that are passed to a routing daemon after the kernel has updated its routing tables.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.6 Error Processing

We first consider the ICMP error messages. A host receives these messages when a datagram that it sent cannot successfully be delivered to its destination. The intended destination host or an intermediate router generates the error message and returns it to the originating system. [Figure 11.17](#) illustrates the format of the various ICMP error messages.

**Figure 11.17. ICMP error messages (icmp\_omitted).**

unreachable time exceeded source quench	type code cksum	void (must be 0)	ip (IP header and data from bad packet) 4 bytes
need fragmentation	type code cksum	pmvoid (must be 0) nextmtu 2 bytes	ip (IP header and data from bad packet) 2 bytes
parameter problem	type code 1 2 bytes	cksum pptr 1 3 bytes	ip (IP header and data from bad packet) at least 28 bytes

The code in Figure 11.18 is from the switch shown in Figure 11.15.

## Figure 11.18. icmp\_input function: error messages.

---

```

176     case ICMP_UNREACH:
177         switch (code) {
178             case ICMP_UNREACH_NET:
179             case ICMP_UNREACH_HOST:
180             case ICMP_UNREACH_PROTOCOL:
181             case ICMP_UNREACH_PORT:
182             case ICMP_UNREACH_SRCFAIL:
183                 code += PRC_UNREACH_NET;
184                 break;
185
186             case ICMP_UNREACH_NEEDFRAG:
187                 code = PRC_MSGSIZE;
188                 break;
189
190             case ICMP_UNREACH_NET_UNKNOWN:
191             case ICMP_UNREACH_NET_PROHIB:
192             case ICMP_UNREACH_TOSNET:
193                 code = PRC_UNREACH_NET;
194                 break;
195
196             case ICMP_UNREACH_HOST_UNKNOWN:
197             case ICMP_UNREACH_ISOLATED:
198             case ICMP_UNREACH_HOST_PROHIB:
199             case ICMP_UNREACH_TOSHOST:
200                 code = PRC_UNREACH_HOST;
201                 break;
202
203             default:
204                 goto badcode;
205             }
206             goto deliver;
207
208         case ICMP_TIMXCEED:
209             if (code > 1)
210                 goto badcode;
211             code += PRC_TIMXCEED_INTRANS;
212             goto deliver;

```

```

208     case ICMP_PARAMPROB:
209         if (code > 1)
210             goto badcode;
211         code = PRC_PARAMPROB;
212         goto deliver;
213     case ICMP_SOURCEQUENCH:
214         if (code)
215             goto badcode;
216         code = PRC_QUENCH;
217     deliver:
218     /*
219      * Problem with datagram; advise higher level routines.
220      */
221     if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp) ||
222         icp->icmp_ip.ip_hl < (sizeof(struct ip) >> 2)) {
223         icmpstat.icps_badlen++;
224         goto freeit;
225     }
226     ntohs(icp->icmp_ip.ip_len);
227     icmpsrc.sin_addr = icp->icmp_ip.ip_dst;
228     if (ctlfunc = inetsw[ip_protox[icp->icmp_ip.ip_p]].pr_ctlinput)
229         (*ctlfunc) (code, (struct sockaddr *) &icmpsrc,
230                     &icp->icmp_ip);
231     break;
232     badcode:
233     icmpstat.icps_badcode++;
234     break;

```

ip\_icmp.c

## 176-216

The processing of ICMP errors is minimal since responsibility for responding to ICMP errors lies primarily with the transport protocols. `icmp_input` maps `icmp_type` and `icmp_code` to a set of protocol-independent error codes represented by the `PRC_` constants. There is an implied ordering of the `PRC_` constants that matches the ICMP code values. This explains why `code` is incremented by a `PRC_` constant.

If the type and code are recognized,

icmp\_input jumps to deliver. If the type and code are not recognized, icmp\_input jumps to badcode.

217-225

If the message length is incorrect for the error being reported, icps\_badlen is incremented and the message discarded. Net/3 always discards invalid ICMP messages, without generating an ICMP error about the invalid message. This prevent an infinite sequence of error messages from forming between two faulty implementations.

226-231

icmp\_input calls the pr\_ctlinput function of the transport protocol that created the *original* IP datagram by demultiplexing the incoming packets to the correct transport protocol based on ip\_p from the original datagram. pr\_ctlinput (if it is defined for the protocol) is passed the error code (code), the destination of the original IP datagram (icmpsdc), and a pointer to the invalid datagram (icmp\_ip). We discuss

these errors with Figures 23.31 and 27.12.

232-234

icps\_badcode is incremented and control breaks out of the switch statement.

### Figure 11.19. The protocol-independent error codes.

Constant	Description
<i>PRC_HOSTDEAD</i>	host appears to be down
<i>PRC_IFDOWN</i>	network interface shut down
<i>PRC_MSGSIZE</i>	invalid message size
<i>PRC_PARAMPROB</i>	header incorrect
<i>PRC_QUENCH</i>	someone said to slow down
<i>PRC_QUENCH2</i>	congestion bit says slow down
<i>PRC_REDIRECT_HOST</i>	host routing redirect
<i>PRC_REDIRECT_NET</i>	network routing redirect
<i>PRC_REDIRECT_TOSHOST</i>	redirect for TOS and host
<i>PRC_REDIRECT_TOSNET</i>	redirect for TOS and network
<i>PRC_ROUTEDEAD</i>	select new route if possible
<i>PRC_TIMXCEED_INTRANS</i>	packet lifetime expired in transit
<i>PRC_TIMXCEED_REASS</i>	fragment lifetime expired during reassembly
<i>PRC_UNREACH_HOST</i>	no route available to host
<i>PRC_UNREACH_NET</i>	no route available to network
<i>PRC_UNREACH_PORT</i>	destination says port is not active
<i>PRC_UNREACH_PROTOCOL</i>	destination says protocol is not available
<i>PRC_UNREACH_SRCFAIL</i>	source route failed

While the PRC\_ constants are ostensibly protocol independent, they are primarily based on the Internet protocols. This results in some loss of specificity when a protocol outside the

# Internet domain maps its errors to the PRC\_ constants.

---

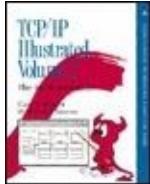
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.7 Request Processing

Net/3 responds to properly formatted ICMP request messages but passes invalid ICMP request messages to `rip_input`. We show in [Chapter 32](#) how ICMP request messages may be generated by an application process.

Most ICMP request messages received by Net/3 generate a reply message, except the router advertisement message. To avoid allocation of a new mbuf for the reply, `icmp_input` converts the mbuf containing the incoming request to the reply and returns it to the sender. We discuss each request separately.

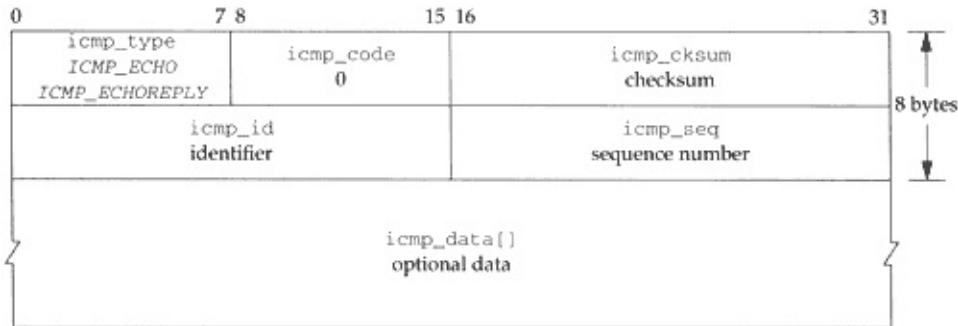
## **Echo Query: ICMP\_ECHO and ICMP\_ECHOREPLY**

For all its simplicity, an ICMP echo request and reply is arguably the single most powerful diagnostic tool available to a network administrator. Sending an ICMP echo request is called *pinging* a host, a reference to the ping program that most systems provide for manually sending ICMP echo requests. Chapter 7 of Volume 1 discusses ping in detail.

The program ping is named after sonar pings used to locate objects by listening for the echo generated as the ping is reflected by the other objects. Volume 1 incorrectly described the name as standing for Packet InterNet Groper.

[\*\*Figure 11.20\*\*](#) shows the structure of an ICMP echo and reply message.

**Figure 11.20. ICMP echo request and reply.**



icmp\_code is always 0. icmp\_id and icmp\_seq are set by the sender of the request and returned without modification in the reply. The source system can match requests and replies with these fields. Any data that arrives in icmp\_data is also reflected. [Figure 11.21](#) shows the ICMP echo processing and also the common code in icmp\_input that implements the reflection of ICMP requests.

**Figure 11.21. icmp\_input function: echo request and reply.**

---

```

235     case ICMP_ECHO:
236         icp->icmp_type = ICMP_ECHOREPLY;
237         goto reflect;

                                         ip_icmp.c

                                         /* other ICMP request processing */

277     reflect:
278         ip->ip_len += hlen;      /* since ip_input deducts this */
279         icmpstat.icps_reflect++;
280         icmpstat.icps_outhist[icp->icmp_type]++;
281         icmp_reflect(m);
282         return;
                                         ip_icmp.c

```

235-237

icmp\_input converts an echo request into an echo reply by changing icmp\_type to ICMP\_ECHOREPLY and jumping to reflect to send the reply.

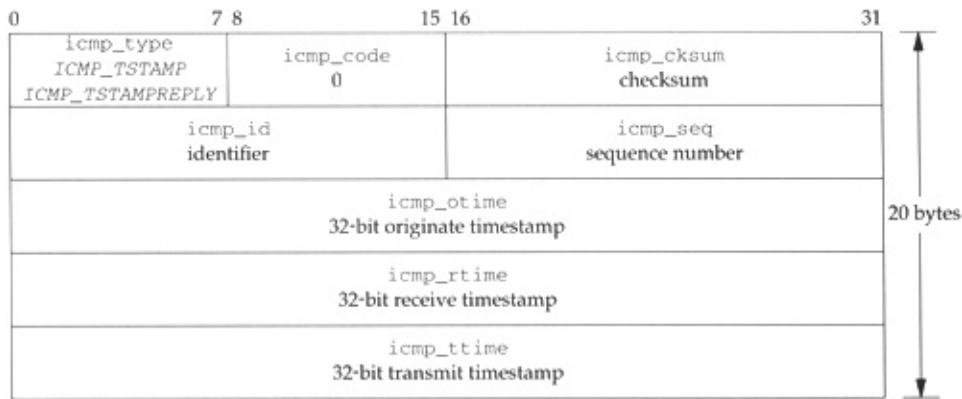
277-282

After constructing the reply for each ICMP request, icmp\_input executes the code at reflect. The correct datagram length is restored, the number of requests and the type of ICMP messages are counted in icps\_reflect and icps\_outhist[], and icmp\_reflect ([Section 11.12](#)) sends the reply back to the requestor.

## Timestamp Query: ICMP\_TSTAMP and ICMP\_TSTAMPREPLY

The ICMP timestamp message is illustrated in [Figure 11.22](#).

**Figure 11.22. ICMP timestamp request and reply.**



`icmp_code` is always 0. `icmp_id` and `icmp_seq` serve the same purpose as those in the ICMP echo messages. The sender of the request sets `icmp_otime` (the time the request originated); `icmp_rtime` (the time the request was received) and `icmp_ttime` (the time the reply was transmitted) are set by the sender of the reply. All times are in milliseconds since midnight UTC; the high-order bit is set if the time value is recorded in nonstandard units, as with the IP timestamp option.

[Figure 11.23](#) shows the code that implements the timestamp messages.

**Figure 11.23. `icmp_input` function: timestamp request and reply.**

```
238     case ICMP_TSTAMP:                                ip_icmp.c
239         if (icmplen < ICMP_TSLEN) {
240             icmpstat.icps_badlen++;
241             break;
242         }
243         icp->icmp_type = ICMP_TSTAMPREPLY;
244         icp->icmp_rtime = iptime();
245         icp->icmp_ttime = icp->icmp_rtime; /* bogus, do later! */
246         goto reflect;
```

## 238-246

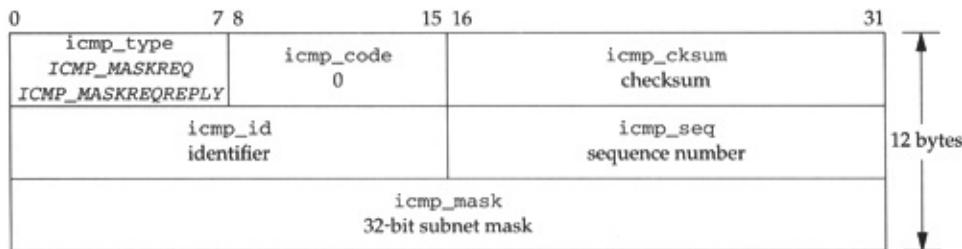
icmp\_input responds to an ICMP timestamp request by changing icmp\_type to ICMP\_TSTAMPREPLY, recording the current time in icmp\_rtime and icmp\_ttime, and jumping to reflect to send the reply.

It is difficult to set icmp\_rtime and icmp\_ttime accurately. When the system executes this code, the message may have already waited on the IP input queue to be processed and icmp\_rtime is set too late. Likewise, the datagram still requires processing and may be delayed in the transmit queue of the network interface so icmp\_ttime is set too early here. To set the timestamps closer to the true receive and transmit times would require modifying the interface drivers for every network to understand ICMP messages (Exercise 11.8).

## Address Mask Query: ICMP\_MASKREQ and ICMP\_MASKREPLY

The ICMP address mask request and reply are illustrated in Figure 11.24.

**Figure 11.24. ICMP address mask request and reply.**



RFC 950 [[Mogul and Postel 1985](#)] added the address mask messages to the original ICMP specification. They enable a system to discover the subnet mask in use on a network.

RFC 1122 forbids sending mask replies unless a system has been explicitly configured as an authoritative agent for address masks. This prevents a system from sharing an incorrect address mask with every system that sends a request.

Without administrative authority to respond, a system should ignore address mask requests.

If the global integer `icmpmaskrepl` is nonzero, Net/3 responds to address mask requests. The default value is 0 and can be changed by `icmp_sysctl` through the `sysctl(8)` program ([Section 11.14](#)).

In Net/2 systems there was no mechanism to control the reply to address mask requests. As a result, it is very important to configure Net/2 interfaces with the correct address mask; the information is shared with any system on the network that sends an address mask request.

The address mask message processing is shown in [Figure 11.25](#).

**Figure 11.25. `icmp_input` function:  
address mask request and reply.**

---

```

247     case ICMP_MASKREQ:
248 #define satosin(sa) ((struct sockaddr_in *) (sa))
249     if (icmpmaskrepl == 0)
250         break;
251     /*
252      * We are not able to respond with all ones broadcast
253      * unless we receive it over a point-to-point interface.
254      */
255     if (icmplen < ICMP_MASKLEN)
256         break;
257     switch (ip->ip_dst.s_addr) {
258     case INADDR_BROADCAST:
259     case INADDR_ANY:
260         icmpdst.sin_addr = ip->ip_src;
261         break;
262     default:
263         icmpdst.sin_addr = ip->ip_dst;
264     }
265     ia = (struct in_ifaddr *) ifaof_ifpforaddr(
266             (struct sockaddr *) &icmpdst, m->m_pkthdr.recvif);
267     if (ia == 0)
268         break;
269     icp->icmp_type = ICMP_MASKREPLY;
270     icp->icmp_mask = ia->ia_sockmask.sin_addr.s_addr;
271     if (ip->ip_src.s_addr == 0) {
272         if (ia->ia_ifp->if_flags & IFF_BROADCAST)
273             ip->ip_src = satosin(&ia->ia_broadaddr)->sin_addr;
274         else if (ia->ia_ifp->if_flags & IFF_POINTOPOINT)
275             ip->ip_src = satosin(&ia->ia_dstaddr)->sin_addr;
276     }

```

---

— ip\_icmp.c

## 247-256

If the system is not configured to respond to mask requests, or if the request is too short, this code breaks out of the switch and passes the message to rip\_input (Figure 11.15).

Net/3 fails to increment icps\_badlen here. It does increment icps\_badlen for all other ICMP length errors.

## Select subnet mask

257-267

If the request was sent to 0.0.0.0 or 255.255.255.255, the source address is saved in icmpdst where it is used by ifaof\_ifpforaddr to locate the in\_ifaddr structure on the same network as the source address. If the source address is 0.0.0.0 or 255.255.255.255, ifaof\_ifpforaddr returns a pointer to the first IP address associated with the receiving interface.

The default case (for unicast or directed broadcasts) saves the destination address for ifaof\_ifpforaddr.

## Convert to reply

269-270

The request is converted into a reply by changing icmp\_type and by copying the selected subnet mask, ia\_sockmask, into icmp\_mask.

## Select destination address

271-276

If the source address of the request is all 0s ("this host on this net," which can be used only as a source address during bootstrap, RFC 1122), then the source does not know its own address and Net/3 must broadcast the reply so the source system can receive the message. In this case, the destination for the reply is `ia_broadaddr` or `ia_dstaddr` if the receiving interface is on a broadcast or point-to-point network, respectively. `icmp_input` puts the destination address for the reply in `ip_src` since the code at `reflect` ([Figure 11.21](#)) calls `icmp_reflect`, which reverses the source and destination addresses. The addresses of a unicast request remain unchanged.

## Information Query: ICMP\_IREQ and ICMP\_IREQREPLY

The ICMP information messages are obsolete. They were intended to allow a host to discover the number of an attached IP network by broadcasting a request with

0s in the network portion of the source and destination address fields. A host responding to the request would return a message with the appropriate network numbers filled in. Some other method was required for a host to discover the host portion of the address.

RFC 1122 recommends that a host not implement the ICMP information messages because RARP (RFC 903 [[Finlayson et al. 1984](#)]), and BOOTP (RFC 951 [[Croft and Gilmore 1985](#)]) are better suited for discovering addresses. A new protocol, the Dynamic Host Configuration Protocol (DHCP), described in RFC 1541 [[Droms 1993](#)], will probably replace and augment the capabilities of BOOTP. It is currently a proposed standard.

Net/2 did respond to ICMP information request messages, but Net/3 passes them on to `rip_input`.

## Router Discovery: `icmp_routeradvert` and `icmp_routersolicit`

RFC 1256 defines the ICMP router discovery messages. The Net/3 kernel does not process these messages directly but instead passes them, by `rip_input`, to a user-level daemon, which sends and responds to the messages.

Section 9.6 of Volume 1 discusses the design and operation of these messages.

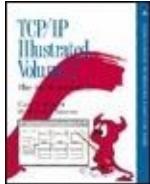
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



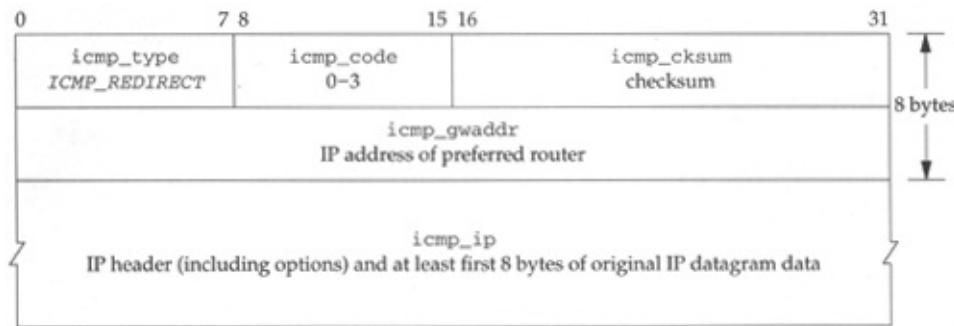
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.8 Redirect Processing

Figure 11.26 shows the format of ICMP redirect messages.

**Figure 11.26. ICMP redirect message.**



The last case to discuss in `icmp_input` is **ICMP\_REDIRECT**. As discussed in Section

**8.5**, a redirect message arrives when a packet is sent to the wrong router. The router forwards the packet to the correct router and sends back a ICMP redirect message, which the system incorporates into its routing tables.

Figure 11.27 shows the code executed by icmp\_input to process redirect messages.

## Figure 11.27. icmp\_input function: redirect messages.

```
283     case ICMP_REDIRECT:
284         if (code > 3)
285             goto badcode;
286         if (icmplen < ICMP_ADVLENMIN || icmplen < ICMP_ADVLEN(icp) ||
287             icp->icmp_ip.ip_hl < (sizeof(struct ip) >> 2)) {
288             icmpstat.icps_badlen++;
289             break;
290         }
291         /*
292          * Short circuit routing redirects to force
293          * immediate change in the kernel's routing
294          * tables. The message is also handed to anyone
295          * listening on a raw socket (e.g. the routing
296          * daemon for use in updating its tables).
297          */
298         icmpgw.sin_addr = ip->ip_src;
299         icmpdst.sin_addr = icp->icmp_gwaddr;
300         icmpsrt.sin_addr = icp->icmp_ip.ip_dst;
301         rtredirect((struct sockaddr *) &icmpsrt,
302                     (struct sockaddr *) &icmpdst,
303                     (struct sockaddr *) 0, RTF_GATEWAY | RTF_HOST,
304                     (struct sockaddr *) &icmpgw, (struct rtentry **) 0);
305         pfctlinput(PRC_REDIRECT_HOST, (struct sockaddr *) &icmpsrt);
306         break;
```

## Validate

283-290

icmp\_input jumps to badcode ([Figure 11.18](#), line 232) if the redirect message includes an unrecognized ICMP code, and drops out of the switch if the message has an invalid length or if the enclosed IP packet has an invalid header length. [Figure 11.16](#) showed that 36 (ICMP\_ADVLENMIN) is the minimum size of an ICMP error message, and ICMP\_ADVLEN (icp) is the minimum size of an ICMP error message including any IP options that may be in the packet pointed to by icp.

291-300

icmp\_input assigns to the static structures icmpgw, icmpdst, and icmpsfc, the source address of the redirect message (the gateway that sent the message), the recommended router for the original packet (the first-hop destination), and the final destination of the original packet.

Here, icmpsfc does not contain a source address; it is a convenient location for holding the destination address instead

of declaring another sockaddr structure.

## Update routes

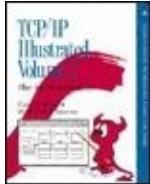
### 301-306

Net/3 follows RFC 1122 recommendations and treats a network redirect and a host redirect identically. The redirect information is passed to rtredirect, which updates the routing tables. The redirected destination (saved in icmpsrt) is passed to pfctlinput, which informs all the protocol domains about the redirect ([Section 7.7](#)). This gives the protocols an opportunity to invalidate any route caches to the destination.

According to RFC 1122, network redirects should be treated as host redirects since they may provide incorrect routing information when the destination network is subnetted. In fact, RFC 1009 requires routers *not* to send network redirects when the network is subnetted. Unfortunately, many routers violate this requirement.

Net/3 never sends network redirects.

ICMP redirect messages are a fundamental part of the IP routing architecture. While classified as an error message, redirect messages appear during normal operations on any network with more than a single router. [Chapter 18](#) covers IP routing issues in more detail.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.9 Reply Processing

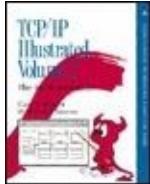
The kernel does not process any of the ICMP reply messages. ICMP requests are generated by processes, never by the kernel, so the kernel passes any replies that it receives to processes waiting for ICMP messages. In addition, the ICMP router discovery messages are passed to `rip_input`.

**Figure 11.28. `icmp_input` function: reply messages.**

```
307     /*
308      * No kernel processing for the following;
309      * just fall through to send to raw listener.
310      */
311     case ICMP_ECHOREPLY:
312     case ICMP_ROUTERADVERT:
313     case ICMP_ROUTERSOLICIT:
314     case ICMP_TSTAMPREPLY:
315     case ICMP_IREQREPLY:
316     case ICMP_MASKREPLY:
317     default:
318         break;
319     }
320 raw:
321     rip_input(m);
322     return;
```

## 307-322

No actions are required by the kernel for ICMP reply messages, so execution continues after the switch statement at raw. Note that the default case for the switch statement (unrecognized ICMP messages) also passes control to the code at raw.



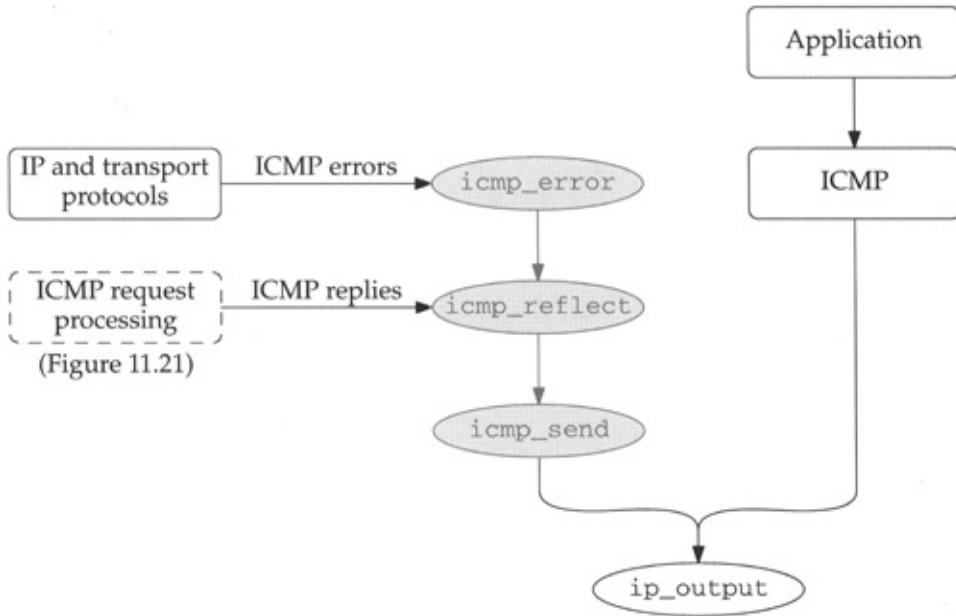
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

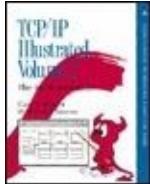
## Chapter 11. ICMP: Internet Control Message Protocol

### 11.10 Output Processing

Outgoing ICMP messages are generated in several ways. We saw in [Chapter 8](#) that IP calls `icmp_error` to generate and send ICMP error messages. ICMP reply messages are sent by `icmp_reflect`, and it is possible for a process to generate ICMP messages through the raw ICMP protocol. [Figure 11.29](#) shows how these functions relate to ICMP output processing.

**Figure 11.29. ICMP output processing.**





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.11 icmp\_error Function

The `icmp_error` function constructs an ICMP error message at the request of IP or the transport protocols and passes it to `icmp_reflect`, where it is returned to the source of the invalid datagram. The function is shown in three parts:

- validate the message ([Figure 11.30](#)),
- construct the header ([Figure 11.32](#)),  
and
- include the original datagram ([Figure 11.33](#)).

## Figure 11.30. icmp\_error function: validation.

```
ip_icmp.c
46 void
47 icmp_error(n, type, code, dest, destifp)
48 struct mbuf *n;
49 int type, code;
50 n_long dest;
51 struct ifnet *destifp;
52 {
53     struct ip *oip = mtod(n, struct ip *), *nip;
54     unsigned oiplen = oip->ip_hl << 2;
55     struct icmp *icmp;
56     struct mbuf *m;
57     unsigned icmplen;
58     if (type != ICMP_REDIRECT)
59         icmpstat.icps_error++;
60     /*
61      * Don't send error if not the first fragment of message.
62      * Don't error if the old packet protocol was ICMP
63      * error message, only known informational types.
64      */
65     if (oip->ip_off & ~(IP_MF | IP_DF))
66         goto freeit;
67     if (oip->ip_p == IPPROTO_ICMP && type != ICMP_REDIRECT &&
68         n->m_len >= oiplen + ICMP_MINLEN &&
69         !ICMP_INFOTYPE((struct icmp *) ((caddr_t) oip + oiplen))->icmp_type)) {
70         icmpstat.icps_olddicmp++;
71         goto freeit;
72     }
73     /* Don't send error in response to a multicast or broadcast packet */
74     if (n->m_flags & (M_BCAST | M_MCAST))
75         goto freeit;
```

46-57

The arguments are: n, a pointer to an mbuf chain containing the invalid datagram; type and code, the ICMP error type and code values; dest, the next-hop router address included in ICMP redirect messages; and destifp, a pointer to the outgoing interface for the original IP packet. mtod converts the mbuf pointer n to oip, a pointer to the ip structure in the

mbuf. The length in bytes of the original IP header is kept in oiplen.

## 58-75

All ICMP errors except redirect messages are counted in icps\_error. Net/3 does not consider redirect messages as errors and icps\_error is not an SNMP variable.

icmp\_error discards the invalid datagram, oip, and does not send an error message if:

- some bits of ip\_off, except those represented by IP\_MF and IP\_DF, are nonzero ([Exercise 11.10](#)). This indicates that oip is not the first fragment of a datagram and that ICMP must not generate error messages for trailing fragments of a datagram.
- the invalid datagram is itself an ICMP error message. ICMP\_INFOTYPE returns true if icmp\_type is an ICMP request or response type and false if it is an error type. This rule avoids creating an infinite sequence of errors about errors.

Net/3 does not consider ICMP redirect messages errors, although RFC 1122 does.

- the datagram arrived as a link-layer broadcast or multicast (indicated by the M\_BCAST and M\_MCAST flags).

ICMP error messages must not be sent in two other circumstances:

- The datagram was sent to an IP broadcast or IP multicast address.
- The datagram's source address is not a unicast IP address (i.e., the source address is a 0 address, a loopback address, a broadcast address, a multicast address, or a class E address)

Net/3 fails to check for the first case. The second case is addressed by the icmp\_reflect function ([Section 11.12](#)).

Interestingly, the Deering multicast extensions to Net/2 do discard datagrams of the first type. Since the Net/3 multicast code was derived from the Deering multicast extensions, it

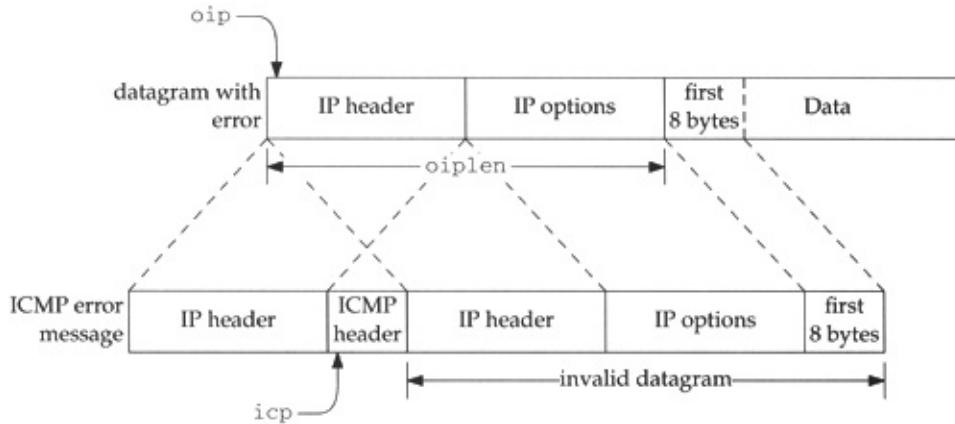
appears the test was removed.

These restrictions attempt to prevent a single broadcast datagram with an error from triggering ICMP error messages from every host on the network. These *broadcast storms* can disrupt communication on a network for an extended period of time as all the hosts attempt to send an error message simultaneously.

These rules apply to ICMP error messages but not to ICMP replies. As RFCs 1122 and 1127 discuss, responding to broadcast requests is allowed but neither recommended nor discouraged. Net/3 responds only to broadcast requests with a unicast source address, since ip\_output will drop ICMP messages returned to a broadcast address ([Figure 11.39](#)).

[Figure 11.31](#) illustrates the construction of an ICMP error message.

**Figure 11.31. The construction of an ICMP error message.**



The code in Figure 11.32 builds the error message.

## Figure 11.32. icmp\_error function: message header construction.

---

```

76  /*
77   * First, formulate icmp message
78   */
79  m = m_gethdr(M_DONTWAIT, MT_HEADER);
80  if (m == NULL)
81      goto freeit;
82  icmplen = oiplen + min(8, oip->ip_len);
83  m->m_len = icmplen + ICMP_MINLEN;
84  MH_ALIGN(m, m->m_len);
85  icp = mtod(m, struct icmp *);
86  if ((u_int) type > ICMP_MAXTYPE)
87      panic("icmp_error");
88  icmpstat.icps_outhist[type]++;
89  icp->icmp_type = type;
90  if (type == ICMP_REDIRECT)
91      icp->icmp_gwaddr.s_addr = dest;
92  else {
93      icp->icmp_void = 0;
94      /*
95       * The following assignments assume an overlay with the
96       * zeroed icmp_void field.
97       */
98      if (type == ICMP_PARAMPROB) {
99          icp->icmp_pptr = code;
100         code = 0;
101     } else if (type == ICMP_UNREACH &&
102                code == ICMP_UNREACH_NEEDFRAG && destifp) {
103         icp->icmp_nextmtu = htons(destifp->if_mtu);
104     }
105 }
106 icp->icmp_code = code;

```

---

## 76-106

icmp\_error constructs the ICMP message header in the following way:

- m\_gethdr allocates a new packet header mbuf. MH\_ALIGN positions the mbuf's data pointer so that the ICMP header, the IP header (and options) of the invalid datagram, and up to 8 bytes of the invalid datagram's data are located at the end of the mbuf.
- icmp\_type, icmp\_code, icmp\_gwaddr (for redirects), icmp\_pptr (for parameter problems), and icmp\_nextmtu (for the fragmentation required message) are initialized. The icmp\_nextmtu field implements the extension to the fragmentation required message described in RFC 1191. Section 24.2 of Volume 1 describes the *path MTU discovery* algorithm, which relies on this message.

Once the ICMP header has been constructed, a portion of the original datagram must be attached to the header,

as shown in Figure 11.33.

## Figure 11.33. icmp\_error function: including the original datagram.

```
107     bcopy((caddr_t) oip, (caddr_t) & icp->icmp_ip, icmplen);           ip_icmp.c
108     nip = &icp->icmp_ip;
109     nip->ip_len = htons((u_short) (nip->ip_len + oiplen));
110
111     /*
112      * Now, copy old ip header (without options)
113      * in front of icmp message.
114      */
115     if (m->m_data - sizeof(struct ip) < m->m_pktdat)
116         panic("icmp len");
117     m->m_data -= sizeof(struct ip);
118     m->m_len += sizeof(struct ip);
119     m->m_pkthdr.len = m->m_len;
120     m->m_pkthdr.rcvif = n->m_pkthdr.rcvif;
121     nip = mtod(m, struct ip *);
122     bcopy((caddr_t) oip, (caddr_t) nip, sizeof(struct ip));
123     nip->ip_len = m->m_len;
124     nip->ip_hl = sizeof(struct ip) >> 2;
125     nip->ip_p = IPPROTO_ICMP;
126     nip->ip_tos = 0;
127     icmp_reflect(m);
128
129 }                                         ip_icmp.c
```

107-125

The IP header, options, and data (a total of icmplen bytes) are copied from the invalid datagram into the ICMP error message. Also, the header length is added back into the invalid datagram's ip\_len.

In udp\_usrreq, UDP also adds the header length back into the invalid

datagram's ip\_len. The result is an ICMP message with an incorrect datagram length in the IP header of the invalid packet. The authors found that many systems based on Net/2 code have this bug. Net/1 systems do not have this problem.

Since MH\_ALIGN located the ICMP message at the end of the mbuf, there should be enough room to prepend an IP header at the front. The IP header (excluding options) is copied from the invalid datagram to the front of the ICMP message.

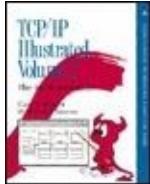
The Net/2 release included a bug in this portion of the code: the last bcopy in the function moved oiplen bytes, which includes the options from the invalid datagram. Only the standard header without options should be copied.

The IP header is completed by restoring the correct datagram length (ip\_len), header length (ip\_hl), and protocol (ip\_p), and clearing the TOS field (ip\_tos).

RFCs 792 and 1122 recommend that the TOS field be set to 0 for ICMP messages.

## 126-129

The completed message is passed to icmp\_reflect, where it is sent back to the source host. The invalid datagram is discarded.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.12 icmp\_reflect Function

icmp\_reflect sends ICMP replies and errors back to the source of the request or back to the source of the invalid datagram. It is important to remember that icmp\_reflect reverses the source and destination addresses in the datagram before sending it. The rules regarding source and destination addresses of ICMP messages are complex. [Figure 11.34](#) summarizes the actions of several functions in this area.

**Figure 11.34. ICMP discard and address summary.**

Function	Summary
icmp_input	Replace an all-0s source address in address mask requests with the broadcast or destination address of the receiving interface.
icmp_error	Discard error messages caused by datagrams sent as link-level broadcasts or multicasts. Should discard (but does not) messages caused by datagrams sent to IP broadcast or multicast addresses.
icmp_reflect	Discard messages instead of returning them to a multicast or experimental address.  Convert nonunicast destinations to the address of the receiving interface, which makes the destination address a valid source address for the return message.  Swap the source and destination addresses.
ip_output	Discards outgoing broadcasts at the request of ICMP (i.e., discards errors generated by packets sent to a broadcast address)

We describe the icmp\_reflect function in three parts: source and destination address selection, option construction, and assembly and transmission. [Figure 11.35](#) shows the first part of the function.

### Figure 11.35. icmp\_reflect function: address selection.

---

```

329 void
330 icmp_reflect(m)
331 struct mbuf *m;
332 {
333     struct ip *ip = mtod(m, struct ip *);
334     struct in_ifaddr *ia;
335     struct in_addr t;
336     struct mbuf *opts = 0, *ip_srcroute();
337     int optlen = (ip->ip_hl << 2) - sizeof(struct ip);

338     if (!in_canforward(ip->ip_src) &&
339         ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) !=
340          (IN_LOOPBACKNET << IN_CLASSA_NSHIFT))) {
341         m_freem(m); /* Bad return address */
342         goto done; /* Ip_output() will check for broadcast */
343     }
344     t = ip->ip_dst;
345     ip->ip_dst = ip->ip_src;
346     /*
347      * If the incoming packet was addressed directly to us,
348      * use dst as the src for the reply. Otherwise (broadcast
349      * or anonymous), use the address which corresponds
350      * to the incoming interface.
351      */
352     for (ia = in_ifaddr; ia; ia = ia->ia_next) {
353         if (t.s_addr == IA_SIN(ia)->sin_addr.s_addr)
354             break;
355         if ((ia->ia_ifp->if_flags & IFF_BROADCAST) &&
356             t.s_addr == satosin(&ia->ia_broadaddr)->sin_addr.s_addr)
357             break;
358     }
359     icmpdst.sin_addr = t;
360     if (ia == (struct in_ifaddr *) 0)
361         ia = (struct in_ifaddr *) ifaof_ifpforaddr(
362                                         (struct sockaddr *) &icmpdst, m->m_pkthdr.rcvif);
363     /*
364      * The following happens if the packet was not addressed to us,
365      * and was received on an interface with no IP address.
366      */
367     if (ia == (struct in_ifaddr *) 0)
368         ia = in_ifaddr;
369     t = IA_SIN(ia)->sin_addr;
370     ip->ip_src = t;
371     ip->ip_ttl = MAXTTL;

```

---

ip\_icmp.c

## Set destination address

329-345

icmp\_reflect starts by making a copy of ip\_dst and moving ip\_src, the source of the request or error datagram, to ip\_dst. icmp\_error and icmp\_reflect ensure that

ip\_src is a valid destination address for the error message. ip\_output discards any packets sent to a broadcast address.

## Select source address

346-371

icmp\_reflect selects a source address for the message by searching in\_ifaddr for the interface with a unicast or broadcast address matching the destination address of the original datagram. On a multihomed host, the matching interface may not be the interface on which the datagram was received. If there is no match, the in\_ifaddr structure of the receiving interface is selected or, failing that (the interface may not be configured for IP), the first address in in\_ifaddr. The function sets ip\_src to the selected address and changes ip\_ttl to 255 (MAXTTL) because the error is a new datagram.

RFC 1700 recommends that the TTL field of all IP packets be set to 64. Many systems, however, set the TTL of ICMP

messages to 255 nowadays.

There is a tradeoff associated with TTL values. A small TTL prevents a packet from circulating in a routing loop but may not allow a packet to reach a site far (many hops) away. A large TTL allows packets to reach distant hosts but lets packets circulate in routing loops for a longer period of time.

RFC 1122 *requires* that source route options, and *recommends* that record route and timestamp options, from an incoming echo request or timestamp request, be attached to a reply. The source route must be reversed in the process. RFC 1122 is silent on how these options should be handled on other types of ICMP replies. Net/3 applies these rules to the address mask request, since it calls `icmp_reflect` ([Figure 11.21](#)) after constructing the address mask reply.

The next section of code ([Figure 11.36](#)) constructs the options for the ICMP message.

## Figure 11.36. icmp\_reflect function: option construction.

```
372     if (optlen > 0) {                                     ip_icmp.c
373         u_char *cp;
374         int     opt, cnt;
375         u_int   len;
376
377         /*
378          * Retrieve any source routing from the incoming packet;
379          * add on any record-route or timestamp options.
380          */
381         cp = (u_char *) (ip + 1);
382         if ((opts = ip_srcroute()) == 0 &&
383             (opts = m_gethdr(M_DONTWAIT, MT_HEADER))) {
384             opts->m_len = sizeof(struct in_addr);
385             mtod(opts, struct in_addr *)->s_addr = 0;
386         }
387         if (opts) {
388             for (cnt = optlen; cnt > 0; cnt -= len, cp += len) {
389                 opt = cp[IPOPT_OPTVAL];
390                 if (opt == IPOPT_EOL)
391                     break;
392                 if (opt == IPOPT_NOP)
393                     len = 1;
394                 else {
395                     len = cp[IPOPT_OLEN];
396                     if (len <= 0 || len > cnt)
397                         break;
398                 }
399                 /*
400                  * Should check for overflow, but it "can't happen"
401                  */
402                 if (opt == IPOPT_RR || opt == IPOPT_TS ||
403                     opt == IPOPT_SECURITY) {
404                     bcopy((caddr_t) cp,
405                           mtod(opts, caddr_t) + opts->m_len, len);
406                     opts->m_len += len;
407                 }
408                 /* Terminate & pad, if necessary */
409                 if (cnt = opts->m_len % 4) {
410                     for (; cnt < 4; cnt++) {
411                         *(mtod(opts, caddr_t) + opts->m_len) =
412                             IPOPT_EOL;
413                         opts->m_len++;
414                     }
415                 }
416             }
417         }
418     }
```

## Get reversed source route

372-385

If the incoming datagram did not contain options, control passes to line 430 ([Figure 11.37](#)). The error messages that icmp\_error sends to icmp\_reflect never have IP options, and so the following code applies only to ICMP requests that are converted to replies and passed directly to icmp\_reflect.

## Figure 11.37. icmp\_reflect function: final assembly.

```
417      /*
418       * Now strip out original options by copying rest of first
419       * mbuf's data back, and adjust the IP length.
420       */
421     ip->ip_len -= optlen;
422     ip->ip_hl = sizeof(struct ip) >> 2;
423     m->m_len -= optlen;
424     if (m->m_flags & M_PKTHDR)
425         m->m_pkthdr.len -= optlen;
426     optlen += sizeof(struct ip);
427     bcopy((caddr_t) ip + optlen, (caddr_t) (ip + 1),
428           (unsigned) (m->m_len - sizeof(struct ip)));
429   }
430   m->m_flags &= ~(M_BCAST | M_MCAST);
431   icmp_send(m, opts);
432 done:
433   if (opts)
434     (void) m_free(opts);
435 }
```

cp points to the start of the options for the *reply*. ip\_srcroute reverses and returns any source route option saved when ipintr processed the datagram. If ip\_srcroute returns 0, the request did not contain a

source route option so icmp\_reflect allocates and initializes an mbuf to serve as an empty ipoption structure.

## Add record route and timestamp options

386-416

If opts points to an mbuf, the for loop searches the options from the *original* IP header and appends the record route and timestamp options to the source route returned by ip\_srcroute.

The options in the original header must be removed before the ICMP message can be sent. This is done by the code shown in [Figure 11.37](#).

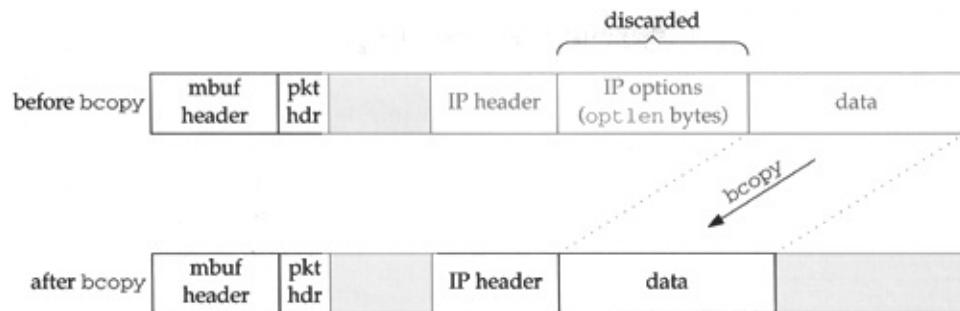
## Remove original options

417-429

icmp\_reflect removes the options from the original request by moving the ICMP message up to the end of the IP header. This is shown in [Figure 11.38](#). The new

options, which are in the mbuf pointed to by opts, are reinserted by ip\_output.

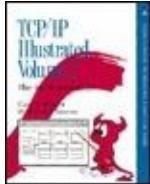
**Figure 11.38. icmp\_reflect: removal of options.**



## Send message and cleanup

430-435

The broadcast and multicast flags are explicitly cleared before passing the message and options to icmp\_send, after which the mbuf containing the options is released.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.13 icmp\_send Function

icmp\_send (Figure 11.39) processes all outgoing ICMP messages and computes the ICMP checksum before passing them to the IP layer.

**Figure 11.39. icmp\_send function.**

```
440 void
441 icmp_send(m, opts)
442 struct mbuf *m;
443 struct mbuf *opts;
444 {
445     struct ip *ip = mtod(m, struct ip *);
446     int     hlen;
447     struct icmp *icp;
448     hlen = ip->ip_hl << 2;
449     m->m_data += hlen;
450     m->m_len -= hlen;
451     icp = mtod(m, struct icmp *);
452     icp->icmp_cksum = 0;
453     icp->icmp_cksum = in_cksum(m, ip->ip_len - hlen);
454     m->m_data -= hlen;
455     m->m_len += hlen;
456     (void) ip_output(m, opts, NULL, 0, NULL);
457 }
```

ip\_icmp.c

## 440-457

As it does when checking the ICMP checksum in icmp\_input, Net/3 adjusts the mbuf data pointer and length to hide the IP header and lets in\_cksum look only at the ICMP message. The computed checksum is placed in the header at icmp\_cksum and the datagram and any options are passed to ip\_output. The ICMP layer does not maintain a route cache, so icmp\_send passes a null pointer to ip\_output instead of a route entry as the third argument. icmp\_send also does not pass any control flags to ip\_output (the fourth argument). In particular, IP\_ALLOWBROADCAST isn't passed, so ip\_output discards any ICMP messages with a broadcast destination address (i.e.,

the original datagram arrived with an invalid source address).

---

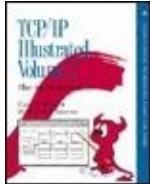
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.14 icmp\_sysctl Function

The `icmp_sysctl` function for IP supports the single option listed in [Figure 11.40](#). The system administrator can modify the option through the `sysctl(8)` program.

**Figure 11.40. icmp\_sysctl parameters.**

sysctl constant	Net/3 variable	Description
<code>ICMPCTL_MASKREPL</code>	<code>icmpmaskrepl</code>	Should system respond to ICMP address mask requests?

[Figure 11.41](#) shows the `icmp_sysctl` function.

## Figure 11.41. icmp\_sysctl function.

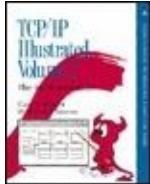
```
ip_icmp.c
467 int
468 icmp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
469 int      *name;
470 u_int     namelen;
471 void     *oldp;
472 size_t   *oldlenp;
473 void     *newp;
474 size_t   newlen;
475 {
476     /* All sysctl names at this level are terminal. */
477     if (namelen != 1)
478         return (ENOTDIR);
479     switch (name[0]) {
480     case ICMPCTL_MASKREPL:
481         return (sysctl_int(oldp, oldlenp, newp, newlen, &icmpmaskrepl));
482     default:
483         return (ENOPROTOOPT);
484     }
485     /* NOTREACHED */
486 }
```

467-478

ENOTDIR is returned if the required ICMP sysctl name is missing.

479-486

There are no options below the ICMP level, so this function calls sysctl\_int to modify icmpmaskrepl or returns ENOPROTOOPT if the option is not recognized.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 11. ICMP: Internet Control Message Protocol

### 11.15 Summary

The ICMP protocol is implemented as a transport layer above IP, but it is tightly integrated with the IP layer. We've seen that the kernel responds directly to ICMP request messages but passes errors and replies to the appropriate transport protocol or application program for processing. The kernel makes immediate changes to the routing tables when an ICMP redirect message arrives but also passes redirects to any waiting processes, typically a routing daemon.

In [Sections 23.9](#) and [27.6](#) we'll see how the UDP and TCP protocols respond to

ICMP error messages, and in [Chapter 32](#) we'll see how a process can generate ICMP requests.

## Exercises

**11.1** What is the source address of an ICMP address mask reply message generated by a request with a destination address of 0.0.0.0?

**11.2** Describe how a link-level broadcast of a packet with a forged unicast source address can interfere with the operation of another host on the network.

**11.3** RFC 1122 suggests that a host should discard an ICMP redirect message if the new first-hop router is on a different subnet from the old first-hop router or if the message came from a router other than the current first-hop router for the final

destination included in the message. Why should this advice be followed?

If the ICMP information request is **11.4** obsolete, why does icmp\_input pass it to rip\_input instead of discarding it?

We pointed out that Net/3 does not convert the offset and length field of an IP packet to network byte **11.5** order before including the packet in an ICMP error message. Why is this inconsequential in the case of the IP offset field?

Describe a situation in which **11.6** ifaof\_ifpforaddr from Figure 11.25 returns a null pointer.

What happens to data included **11.7** after the timestamps in a timestamp query?

Implement the following changes to improve the ICMP timestamp code:

Add a timestamp field to the mbuf packet header. Have the device drivers record the exact time a packet is received in this field and have the ICMP timestamp code copy the value into the icmp\_rtime field.

**11.8**

On output, have the ICMP timestamp code store the byte offset of where in the packet to store the current time in the timestamp field. Modify a device driver to insert the time-stamp right before sending the packet.

**11.9**      Modify icmp\_error to return up to 64 bytes (as does Solaris 2.x) of the original datagram in ICMP error messages.

In [Figure 11.30](#), what happens to a

**11.10** packet that has the high-order bit of ip\_off set?

Why is the return value from  
**11.11** ip\_output discarded in Figure  
11.39?

---

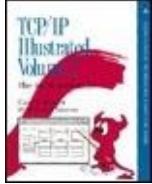
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 12. IP Multicasting

Section 12.1. Introduction

Section 12.2. Code Introduction

Section 12.3. Ethernet Multicast Addresses

Section 12.4. ether\_multi Structure

Section 12.5. Ethernet Multicast Reception

Section 12.6. in\_multi Structure

Section 12.7. ip\_moptions Structure

Section 12.8. Multicast Socket Options

Section 12.9. Multicast TTL Values

Section 12.10. ip\_setmoptions Function

Section 12.11. Joining an IP Multicast Group

Section 12.12. Leaving an IP Multicast

Group

Section 12.13. ip\_getmoptions  
Function

Section 12.14. Multicast Input  
Processing: ipintr Function

Section 12.15. Multicast Output  
Processing: ip\_output Function

Section 12.16. Performance  
Considerations

Section 12.17. Summary

---

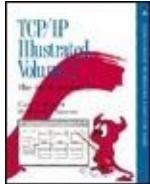
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.1 Introduction

Recall from [Chapter 8](#) that class D IP addresses (224.0.0.0 to 239.255.255.255) do not identify individual interfaces in an internet but instead identify groups of interfaces. For this reason, class D addresses are called *multicast groups*. A datagram with a class D destination address is delivered to every interface in an internet that has *joined* the corresponding multicast group.

Experimental applications on the Internet that take advantage of multicasting include audio and video conferencing applications, resource discovery tools, and shared whiteboards.

Group membership is determined dynamically as interfaces join and leave groups based on requests from processes running on each system. Since group membership is relative to an interface, it is possible for a multihomed host to have different group membership lists for each interface. We'll refer to group membership on a particular interface as an {interface, group} pair.

Group membership on a single network is communicated between systems by the IGMP protocol ([Chapter 13](#)). Multicast routers propagate group membership information using multicast routing protocols ([Chapter 14](#)), such as DVMRP (Distance Vector Multicast Routing Protocol). A standard IP router may support multicast routing, or multicast routing may be handled by a router dedicated to that purpose.

Networks such as Ethernet, token ring, and FDDI directly support hardware multicasting. In Net/3, if an interface supports multicasting, the IFF\_MULTICAST bit is on in if\_flags in the interface's ifnet

structure ([Figure 3.7](#)). We'll use Ethernet to illustrate hardware-supported IP multicasting, since Ethernet is in widespread use and Net/3 includes sample Ethernet drivers. Multicast services are trivially implemented on point-to-point networks such as SLIP and the loopback interface.

IP multicasting services may not be available on a particular interface if the local network does not support hardware-level multicast. RFC 1122 does not prevent the interface layer from providing a software-level multicast service as long as it is transparent to IP.

RFC 1112 [[Deering 1989](#)] describes the host requirements for IP multicasting. There are three levels of conformance:

Level 0 The host cannot send or receive IP multicasts.

Such a host should silently discard any packets it receives with a class D destination address.

Level 1 The host can send but cannot receive IP multicasts.

A host is not required to join an IP multicast group before sending a datagram to the group. A multicast datagram is sent in the same way as a unicast datagram except the destination address is the IP multicast group. The network drivers must recognize this and multicast the datagram on the local network.

Level 2 The host can send and receive IP multicasts.

To receive IP multicasts, the host must be able to join and leave multicast groups and must support IGMP for exchanging group membership information on at least one interface. A multihomed host may support multicasting on a subset of its interfaces.

Net/3 meets the level 2 host requirements and can additionally act as a multicast router. As with unicast IP routing, we assume that the system we are describing is a multicast router and we include the Net/3 multicast routing code in our presentation.

## Well-Known IP Multicast Groups

As with UDP and TCP port numbers, the *Internet Assigned Numbers Authority* (IANA) maintains a list of registered IP multicast groups. The current list can be found in RFC 1700. For more information about the IANA, see RFC 1700. [Figure 12.1](#) shows only some of the well-known groups.

**Figure 12.1. Some registered IP multicast groups.**

Group	Description	Net/3 constant
224.0.0.0	reserved	<i>INADDR_UNSPEC_GROUP</i>
224.0.0.1	all systems on this subnet	<i>INADDR_ALLHOSTS_GROUP</i>
224.0.0.2	all routers on this subnet	
224.0.0.3	unassigned	
224.0.0.4	DVMRP routers	
224.0.0.255	unassigned	<i>INADDR_MAX_LOCAL_GROUP</i>
224.0.1.1	NTP Network Time Protocol	
224.0.1.2	SGI-Dogfight	

The first 256 groups (224.0.0.0 to 224.0.0.255) are reserved for protocols that implement IP unicast and multicast routing mechanisms. Datagrams sent to any of these groups are not forwarded beyond the local network by multicast routers, regardless of the TTL value in the IP header.

RFC 1075 places this requirement only on the 224.0.0.0 and 224.0.0.1 groups but mrouted, the most common multicast routing implementation, restricts the remaining groups as described here. Group 224.0.0.0 (*INADDR\_UNSPEC\_GROUP*) is reserved and group 224.0.0.255 (*INADDR\_MAX\_LOCAL\_GROUP*) marks the last local multicast group.

Every level-2 conforming system is

required to join the 224.0.0.1 (INADDR\_ALLHOSTS\_GROUP) group on all multicast interfaces at system initialization time ([Figure 6.19](#)) and remain a member of the group until the system is shut down. There is no multicast group that corresponds to every interface on an internet.

Imagine if your voice-mail system had the option of sending a message to every voice mailbox in your company. Maybe you have such an option. Do you find it useful? Does it scale to larger companies? Can anyone send to the "all-mailbox" group, or is it restricted?

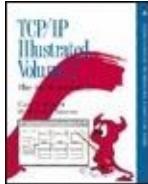
Unicast and multicast routers may join group 224.0.0.2 to communicate with each other. The ICMP router solicitation message and router advertisement messages may be sent to 224.0.0.2 (the all-routers group) and 224.0.0.1 (the all-hosts group), respectively, instead of to the limited broadcast address (255.255.255.255).

The 224.0.0.4 group supports

communication between multicast routers that implement DVMRP. Other groups within the local multicast group range are similarly assigned for other routing protocols.

Beyond the first 256 groups, the remaining groups (224.0.1.0-239.255.255.255) are assigned to various multicast application protocols or remain unassigned. [Figure 12.1](#) lists two examples, the Network Time Protocol (224.0.1.1), and SGI-Dogfight (224.0.1.2).

Throughout this chapter, we note that multicast packets are sent and received by the transport layer on a host. While the multicasting code is not aware of the specific transport protocol that sends and receives multicast datagrams, the only Internet transport protocol that supports multicasting is UDP.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

## 12.2 Code Introduction

The basic multicasting code discussed in this chapter is contained within the same files as the standard IP code. [Figure 12.2](#) lists the files that we examine.

**Figure 12.2. Files discussed in this chapter.**

File	Description
netinet/if_ether.h	Ethernet multicasting structure and macro definitions
netinet/in.h	more Internet multicast structures
netinet/in_var.h	Internet multicast structure and macro definitions
netinet/ip_var.h	IP multicast structures
net/if_ETHERSUBR.C	Ethernet multicast functions
netinet/in.c	group membership functions
netinet/ip_input.c	input multicast processing
netinet/ip_output.c	output multicast processing

## Global Variables

Three new global variables are introduced in this chapter:

**Figure 12.3. Global variables introduced in this chapter.**

Variable	Datatype	Description
ether_ipmulticast_min	u_char []	minimum Ethernet multicast address reserved for IP
ether_ipmulticast_max	u_char []	maximum Ethernet multicast address reserved for IP
ip_mrouted	struct socket *	pointer to socket created by multicast routing daemon

## Statistics

The code in this chapter updates a few of the counters maintained in the global ipstat structure.

**Figure 12.4. Multicast processing statistics.**

ipstat member	Description
ips_forward	#packets forwarded by this system
ips_cantforward	#packets that cannot be forwarded—system is not a router
ips_noroute	#packets that cannot be forwarded because a route is not available

Link-level multicast statistics are collected in the ifnet structure ([Figure 4.5](#)) and may include multicasting of protocols other than IP.

---

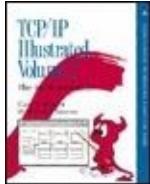
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.3 Ethernet Multicast Addresses

An efficient implementation of IP multicasting requires IP to take advantage of hardware-level multicasting, without which each IP datagram would have to be broadcast to the network and every host would have to examine each datagram and discard those not intended for the host. The hardware filters unwanted datagrams before they reach the IP layer.

For the hardware filter to work, the network interface must convert the IP multicast group destination to a link-layer multicast address recognized by the network hardware. On point-to-point

networks, such as SLIP and the loopback interface, the mapping is implicit since there is only one possible destination. On other networks, such as Ethernet, an explicit mapping function is required. The standard mapping for Ethernet applies to any network that employs 802.3 addressing.

[Figure 4.12](#) illustrated the difference between an Ethernet unicast and multicast address: if the low-order bit of the high-order byte of the Ethernet address is a 1, it is a multicast address; otherwise it is a unicast address. Unicast Ethernet addresses are assigned by the interface's manufacturer, but multicast addresses are assigned dynamically by network protocols.

## IP to Ethernet Multicast Address Mapping

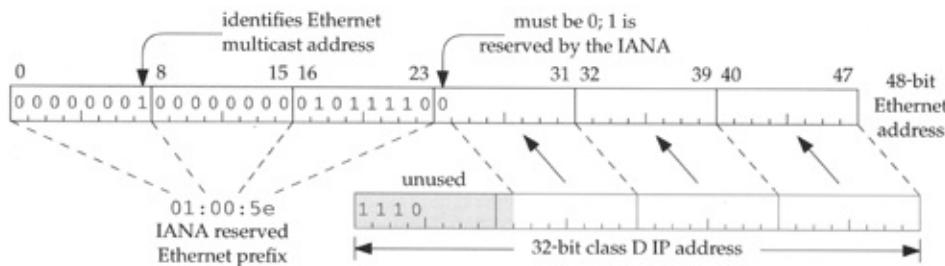
Because Ethernet supports multiple protocols, a method to allocate the multicast addresses and prevent conflicts is needed. Ethernet addresses allocation is

administered by the IEEE. A block of Ethernet multicast addresses is assigned to the IANA by the IEEE to support IP multicasting. The addresses in the block all start with 01:00:5e.

The block of Ethernet unicast addresses starting with 00:00:5e is also assigned to the IANA but remains reserved for future use.

Figure 12.5 illustrates the construction of an Ethernet multicast address from a class D IP address.

### Figure 12.5. Mapping between IP and Ethernet addresses.



The mapping illustrated by Figure 12.5 is a many-to-one mapping. The high-order 9 bits of the class D IP address are not used

when constructing the Ethernet address. 32 IP multicast groups map to a single Ethernet multicast address ([Exercise 12.3](#)). In [Section 12.14](#) we'll see how this affects input processing. [Figure 12.6](#) shows the macro that implements this mapping in Net/3.

**Figure 12.6.  
ETHER\_MAP\_IP\_MULTICAST macro.**

```
61 #define ETHER_MAP_IP_MULTICAST(ipaddr, enaddr) \if_ether.h
62     /* struct in_addr *ipaddr; */ \
63     /* u_char enaddr[6]; */ \
64 { \
65     (enaddr)[0] = 0x01; \
66     (enaddr)[1] = 0x00; \
67     (enaddr)[2] = 0x5e; \
68     (enaddr)[3] = ((u_char *)ipaddr)[1] & 0x7f; \
69     (enaddr)[4] = ((u_char *)ipaddr)[2]; \
70     (enaddr)[5] = ((u_char *)ipaddr)[3]; \
71 } \if_ether.h
```

## IP to Ethernet multicast mapping

61-71

ETHER\_MAP\_IP\_MULTICAST implements the mapping shown in [Figure 12.5](#). ipaddr points to the class D multicast address, and the matching Ethernet address is constructed in enaddr, an array of 6 bytes.

The first 3 bytes of the Ethernet multicast address are 0x01,0x00, and 0x5e followed by a 0 bit and then the low-order 23 bits of the class D IP address.

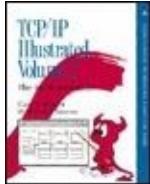
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.4 ether\_multi Structure

For each Ethernet interface, Net/3 maintains a list of Ethernet multicast address ranges to be received by the hardware. This list defines the multicast filtering to be implemented by the device. Because most Ethernet devices are limited in the number of addresses they can selectively receive, the IP layer must be prepared to discard datagrams that pass through the hardware filter. Each address range is stored in an ether\_multi structure:

**Figure 12.7. ether\_multi structure.**

```
147 struct ether_multi {  
148     u_char enm_addrlo[6];      /* low or only address of range */  
149     u_char enm_addrhi[6];      /* high or only address of range */  
150     struct arpcom *enm_ac;    /* back pointer to arpcom */  
151     u_int enm_refcount;       /* no. claims to this addr/range */  
152     struct ether_multi *enm_next; /* ptr to next ether_multi */  
153 };
```

## Ethernet multicast addresses

147-153

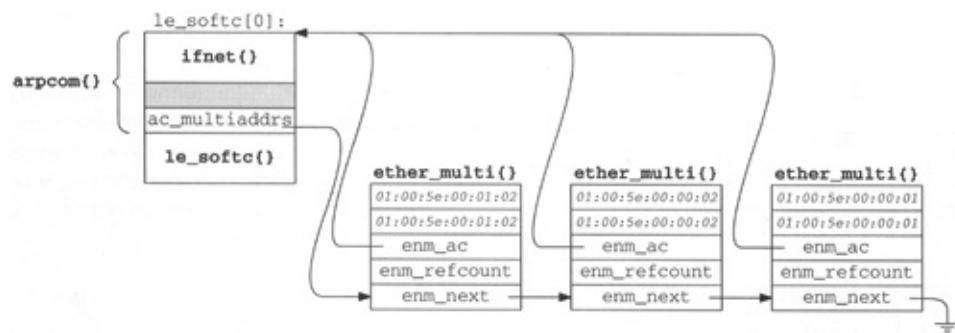
enm\_addrlo and enm\_addrhi specify a range of Ethernet multicast addresses that should be received. A single Ethernet address is specified when enm\_addrlo and enm\_addrhi are the same. The entire list of ether\_multi structures is attached to the arpcom structure of each Ethernet interface ([Figure 3.26](#)). Ethernet multicasting is independent of ARP using the arpcom structure is a matter of convenience, since the structure is already included in every Ethernet interface structure.

We'll see that the start and end of the ranges are always the same since there is no way in Net/3 for a process to specify an address range.

`enm_ac` points back to the `arpcom` structure of the associated interface and `enm_refcount` tracks the usage of the `ether_multi` structure. When the reference count drops to 0, the structure is released. `enm_next` joins the `ether_multi` structures for a single interface into a linked list.

[Figure 12.8](#) shows a list of three `ether_multi` structures attached to `le_softc[0]`, the `ifnet` structure for our sample Ethernet interface.

**Figure 12.8. The LANCE interface with three ether\_multi structures.**



In [Figure 12.8](#) we see that:

- The interface has joined three groups. Most likely they are: 224.0.0.1 (all-hosts), 224.0.0.2 (all-routers), and

224.0.1.2 (SGI-dogfight). Because the Ethernet to IP mapping is a one-to-many mapping, we cannot determine the exact IP multicast groups by examining the resulting Ethernet multicast addresses. The interface may have joined 225.0.0.1, 225.0.0.2, and 226.0.1.2, for example.

- The most recently joined group appears at the front of the list.
- The enm\_ac back-pointer makes it easy to find the beginning of the list and to release an ether\_multi structure, without having to implement a doubly linked list.
- The ether\_multi structures apply to Ethernet devices only. Other multicast devices may have a different multicast implementation.

The ETHER\_LOOKUP\_MULTI macro, shown in [Figure 12.9](#), searches an ether\_multi list for a range of addresses.

**Figure 12.9. ETHER\_LOOKUP\_MULTI**

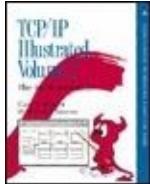
## macro.

```
166 #define ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm) \ if_ether.h
167     /* u_char addrlo[6]; */ \
168     /* u_char addrhi[6]; */ \
169     /* struct arpcom *ac; */ \
170     /* struct ether_multi *enm; */ \
171 { \
172     for ((enm) = (ac)->ac_multiaddrs; \
173         (enm) != NULL && \
174         (bcmpl((enm)->enm_addrlo, (addrlo), 6) != 0 || \
175          bcmpl((enm)->enm_addrhi, (addrhi), 6) != 0); \
176         (enm) = (enm)->enm_next); \
177 } if_ether.h
```

## Ethernet multicast lookups

166-177

addrlo and addrhi specify the search range and ac points to the arpcom structure containing the list to search. The for loop performs a linear search, stopping at the end of the list or when enm\_addrlo and enm\_addrhi both match the supplied addrlo and addrhi addresses. When the loop terminates, enm is null or points to a matching ether\_multi structure.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.5 Ethernet Multicast Reception

After this section, this chapter discusses only IP multicasting, but it is possible in Net/3 to configure the system to receive any Ethernet multicast packet. Although not useful with the IP protocols, other protocol families within the kernel might be prepared to receive these multicasts. Explicit multicast configuration is done by issuing the ioctl commands shown in Figure 12.10.

**Figure 12.10. Multicast ioctl commands.**

Command	Argument	Function	Description
SIOCADDMULTI	struct ifreq *	ifioctl	add multicast address to reception list
SIOCDELMULTI	struct ifreq *	ifioctl	delete multicast address from reception list

These two commands are passed by ifioctl ([Figure 12.11](#)) directly to the device driver for the interface specified in the ifreq structure ([Figure 6.12](#)).

## Figure 12.11. ifioctl function: multicast commands.

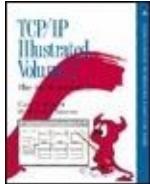
---

```
if.c
440     case SIOCADDMULTI:
441     case SIOCDELMULTI:
442         if (error = suser(p->p_ucred, &p->p_acflag))
443             return (error);
444         if (ifp->if_ioctl == NULL)
445             return (EOPNOTSUPP);
446         return ((*ifp->if_ioctl) (ifp, cmd, data));
if.c
```

---

440-446

If the process does not have superuser privileges, or if the interface does not have an if\_ioctl function, ifioctl returns an error; otherwise the request is passed directly to the device driver.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.6 in\_multi Structure

The Ethernet multicast data structures described in [Section 12.4](#) are not specific to IP; they must support multicast activity by any of the protocol families supported by the kernel. At the network level, IP maintains a list of IP multicast groups associated with each interface.

As a matter of implementation convenience, the IP multicast list is attached to the `in_ifaddr` structure associated with the interface. Recall from [Section 6.5](#) that this structure contains the unicast address for the interface. There is no relationship between the unicast address and the attached multicast group

list other than that they both are associated with the same interface.

This is an artifact of the Net/3 implementation. It is possible for an implementation to support IP multicast groups on an interface that does not accept IP unicast packets.

Each IP multicast {interface, group} pair is described by an `in_multi` structure shown in [Figure 12.12](#).

**Figure 12.12. `in_multi` structure.**

```
111 struct in_multi {
112     struct in_addr inm_addr;      /* IP multicast address */
113     struct ifnet *inm_ifp;        /* back pointer to ifnet */
114     struct in_ifaddr *inm_ia;     /* back pointer to in_ifaddr */
115     u_int    inm_refcount;       /* no. membership claims by sockets */
116     u_int    inm_timer;          /* IGMP membership report timer */
117     struct in_multi *inm_next;   /* ptr to next multicast address */
118 };
```

## IP multicast addresses

111-118

`inm_addr` is a class D multicast address (e.g., 224.0.0.1, the all-hosts group).

`inm_ifp` points back to the `ifnet` structure of the associated interface and `inm_ia` points back to the interface's `in_ifaddr` structure.

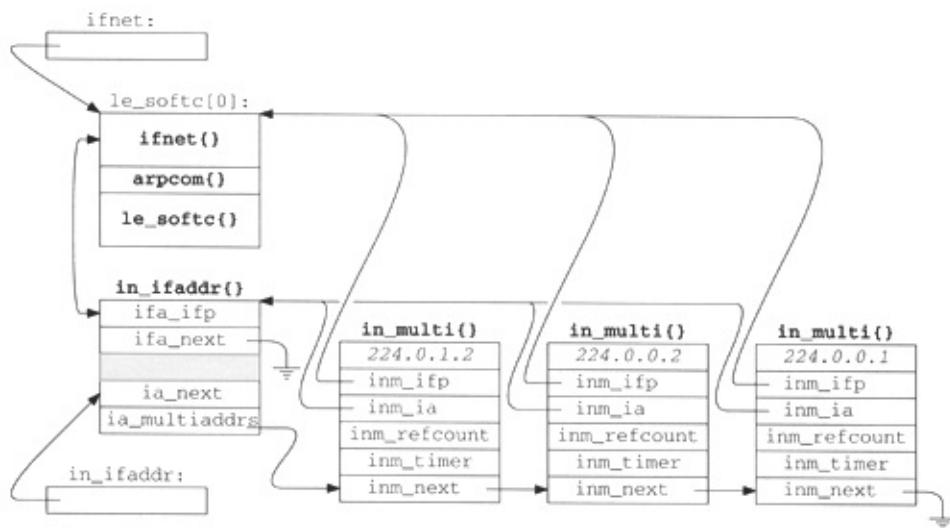
An `in_multi` structure exists only if at least one process on the system has notified the kernel that it wants to receive multicast datagrams for a particular {interface, group} pair. Since multiple processes may elect to receive datagrams sent to a particular pair, `inm_refcount` keeps track of the number of references to the pair. When no more processes are interested in the pair, `inm_refcount` drops to 0 and the structure is released. This action may cause an associated `ether_multi` structure to be released if its reference count also drops to 0.

`inm_timer` is part of the IGMP protocol implementation described in [Chapter 13](#). Finally, `inm_next` points to the next `in_multi` structure in the list.

[Figure 12.13](#) illustrates the relationship between an interface, its IP unicast address, and its IP multicast group list

using the `le_softc[0]` sample interface.

**Figure 12.13. An IP multicast group list for the le interface.**



We've omitted the corresponding `ether_multi` structures for clarity (but see [Figure 12.34](#)). If the system had two Ethernet cards, the second card would be managed through `le_softc[1]` and would have its own multicast group list attached to its `arpcom` structure. The macro [IN\\_LOOKUP\\_MULTI \(Figure 12.14\)](#) searches the IP multicast list for a particular multicast group.

## Figure 12.14. IN\_LOOKUP\_MULTI macro.

```
131 #define IN_LOOKUP_MULTI(addr, ifp, inm) \-----in_var.h
132     /* struct in_addr addr; */ \
133     /* struct ifnet *ifp; */ \
134     /* struct in_multi *inm; */ \
135 { \
136     struct in_ifaddr *ia; \
137 \
138     IFP_TO_IA(ifp, ia); \
139     if (ia == NULL) \
140         (inm) = NULL; \
141     else \
142         for ((inm) = ia->ia_mult��addrs; \
143             (inm) != NULL && (inm)->inm_addr.s_addr != (addr).s_addr; \
144             (inm) = inm->inm_next) \
145             continue; \
146 }-----in_var.h
```

## IP multicast lookups

131-146

IN\_LOOKUP\_MULTI looks for the multicast group addr in the multicast group list associated with interface ifp. IFP\_TO\_IA searches the Internet address list, in\_ifaddr, for the in\_ifaddr structure associated with the interface identified by ifp. If IFP\_TO\_IA finds an interface, the for loop searches its IP multicast list. After the loop, inm is null or points to the matching in\_multi structure.

Top

## Chapter 12. IP Multicasting

---

### 12.7 ip\_moptions Structure

The `ip_moptions` structure contains the multicast options used for multicast output processing. For example, the line

```
error = ip_output(m, inp->inp_opaque,
                   inp->inp_socket,
                   inp->inp_mopti
```

In Chapter 22 we'll see that `inp` points to an `InternetProtocolBlock` structure which associates a PCB with each socket created by a call to `socket`. The `ip_moptions` structure is stored in the `inp_mopti` field of the `InternetProtocolBlock` structure. This means that `ip_output` is called for each outgoing datagram. **Figure 1**

**Figure 12.15.**

```
100 struct ip_moptions {  
101     struct ifnet *imo_multicast_ifp; /* ifp for outgoing multicasts */  
102     u_char imo_multicast_ttl;      /* TTL for outgoing multicasts */  
103     u_char imo_multicast_loop;    /* 1 => hear sends if a member */  
104     u_short imo_num_memberships; /* no. memberships this socket */  
105     struct in_multi *imo_membership[IP_MAX_MEMBERSHIPS];  
106 };
```

ip\_var.h

## Multicast options

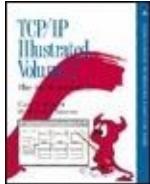
100-106

ip\_output routes outgoing multicast datagrams if imo\_multicast\_ifp is null, through the default [14](#)).

imo\_multicast\_ttl specifies the initial IP TTL value for multicast datagrams to remain on the local network.

If imo\_multicast\_loop is 0, the multicast datagram is discarded at the interface even if the interface is a member of the group. If it is 1, the multicast datagram is looped back to the transmitter from the local interface. This allows a host to receive its own multicast group traffic.

Finally, the integer imo\_num\_memberships and {interface, group} pairs associated with the structure are used to announce membership changes on the locally attached interfaces. The array imo\_membership is a pointer to an in\_multi structure attached to each interface.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.8 Multicast Socket Options

Several IP-level socket options, shown in [Figure 12.16](#), provide process-level access to ip\_moptions structures.

**Figure 12.16. Multicast socket options.**

Command	Argument	Function	Description
<code>IP_MULTICAST_IF</code>	<code>struct in_addr</code>	<code>ip_ctloutput</code>	select default interface for outgoing multicasts
<code>IP_MULTICAST_TTL</code>	<code>u_char</code>	<code>ip_ctloutput</code>	select default TTL for outgoing multicasts
<code>IP_MULTICAST_LOOP</code>	<code>u_char</code>	<code>ip_ctloutput</code>	enable or disable loopback of outgoing multicasts
<code>IP_ADD_MEMBERSHIP</code>	<code>struct ip_mreq</code>	<code>ip_ctloutput</code>	join a multicast group
<code>IP_DROP_MEMBERSHIP</code>	<code>struct ip_mreq</code>	<code>ip_ctloutput</code>	leave a multicast group

In [Figure 8.31](#) we looked at the overall structure of the `ip_ctloutput` function. [Figure 12.17](#) shows the cases relevant to changing and retrieving multicast options.

## Figure 12.17. ip\_ctloutput function: multicast options.

```
448     case PRCO_SETOPT:
449         switch (optname) {
450             /* other set cases */
451
486             case IP_MULTICAST_IF:
487             case IP_MULTICAST_TTL:
488             case IP_MULTICAST_LOOP:
489             case IP_ADD_MEMBERSHIP:
490             case IP_DROP_MEMBERSHIP:
491                 error = ip_setmoptions(optname, &inp->inp_moptions, m);
492                 break;
493                 freeit:
494             default:
495                 error = EINVAL;
496                 break;
497             }
498             if (m)
499                 (void) m_free(m);
500             break;
501
502     case PRCO_GETOPT:
503         switch (optname) {
504             /* other get cases */
505
539             case IP_MULTICAST_IF:
540             case IP_MULTICAST_TTL:
541             case IP_MULTICAST_LOOP:
542             case IP_ADD_MEMBERSHIP:
543             case IP_DROP_MEMBERSHIP:
544                 error = ip_getmoptions(optname, inp->inp_moptions, mp);
545                 break;
546             default:
547                 error = ENOPROTOOPT;
548                 break;
549         }

```

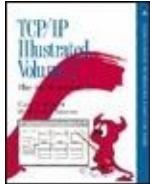
486-491 539-549

All the multicast options are handled through the ip\_setmoptions and ip\_getmoptions functions. The ip\_moptions structure passed by reference to ip\_getmoptions or to ip\_setmoptions is the

one associated with the socket on which the ioctl command was issued.

The error code returned when an option is not recognized is different for the get and set cases. ENOPROTOOPT is the more reasonable choice.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.9 Multicast TTL Values

Multicast TTL values are difficult to understand because they have two purposes. The primary purpose of the TTL value, as with all IP packets, is to limit the lifetime of the packet within an internet and prevent it from circulating indefinitely. The second purpose is to contain packets within a region of the internet specified by administrative boundaries. This administrative region is specified in subjective terms such as "this site," "this company," or "this state," and is relative to the starting point of the packet. The region associated with a multicast packet is called its *scope*.

The standard implementation of RFC 1112 multicasting merges the two concepts of lifetime and scope into the single TTL value in the IP header. In addition to discarding packets when the IP TTL drops to 0, *multicast* routers associate with each interface a TTL threshold that limits multicast transmission on that interface. A packet must have a TTL greater than or equal to the interface's threshold value for it to be transmitted on the interface. Because of this, a multicast packet may be dropped even before its TTL value reaches 0.

Threshold values are assigned by an administrator when configuring a multicast router. These values define the scope of multicast packets. The significance of an initial TTL value for multicast datagrams is defined by the threshold policy used by the administrator and the distance between the source of the datagram and the multicast interfaces.

[Figure 12.18](#) shows the recommended TTL values for various applications as well as recommended threshold values.

**Figure 12.18. TTL values for IP multicast datagrams.**

ip_ttl	Application	Scope
0		same interface
1		same subnet
31	local event video	
32		same site
63	local event audio	
64		same region
95	IETF channel 2 video	
127	IETF channel 1 video	
128		same continent
159	IETF channel 2 audio	
191	IETF channel 1 audio	
223	IETF channel 2 low-rate audio	
255	IETF channel 1 low-rate audio unrestricted in scope	

The first column lists the starting value of ip\_ttl in the IP header. The second column illustrates an application specific use of threshold values ([Casner 1993]). The third column lists the recommended scopes to associate with the TTL values.

For example, an interface that communicates to a network outside the local site would be configured with a multicast threshold of 32. The TTL field of any datagram that starts with a TTL of 32 (or less) is less than 32 when it reaches this interface (there is at least one hop

between the source and the router) and is discarded before the router forwards it to the external network even if the TTL is still greater than 0.

A multicast datagram that starts with a TTL of 128 would pass through site interfaces with a threshold of 32 (as long as it reached the interface within  $128 - 32 = 96$  hops) but would be discarded by intercontinental interfaces with a threshold of 128.

## The MBONE

A subset of routers on the Internet supports IP multicast routing. This multicast backbone is called the *MBONE*, which is described in [Casner 1993]. It exists to support experimentation with IP multicasting in particular with audio and video data streams. In the MBONE, threshold values limit how far various data streams propagate. In Figure 12.18, we see that local event video packets always start with a TTL of 31. An interface with a threshold of 32 always blocks local event

video. At the other end of the scale, IETF channel 1 low-rate audio is restricted only by the inherent IP TTL maximum of 255 hops. It propagates through the entire MBONE. An administrator of a multicast router within the MBONE can select a threshold value to accept or discard MBONE data streams selectively.

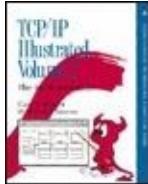
## Expanding-Ring Search

Another use of the multicast TTL is to probe the internet for a resource by varying the initial TTL value of the probe datagram. This technique is called an *expanding-ring search* ([[Boggs 1982](#)]). A datagram with an initial TTL of 0 reaches only a resource on the local system associated with the outgoing interface. A TTL of 1 reaches the resource if it exists on the local subnet. A TTL of 2 reaches resources within two hops of the source. An application increases the TTL exponentially to probe a large internet quickly.

RFC 1546 [[Partridge, Mendez, and](#)

Milliken 1993] describes a related service called *anycasting*. As proposed, anycasting relies on a distinguished set of IP addresses to represent groups of hosts much like multicasting. Unlike multicast addresses, the network is expected to propagate an anycast packet until it is received by at least one host. This simplifies the implementation of an application, which no longer needs to perform expanding-ring searches.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

## 12.10 ip\_setmoptions Function

The bulk of the `ip_setmoptions` function consists of a switch statement to handle each option. [Figure 12.19](#) shows the beginning and end of `ip_setmoptions`. The body of the switch is discussed in the following sections.

**Figure 12.19. `ip_setmoptions` function.**

---

```

650 int ip_setmoptions(optname, imop, m)
651 ip_setmoptions(optname, imop, m)
652 int optname;
653 struct ip_moptions **imop;
654 struct mbuf *m;
655 {
656     int error = 0;
657     u_char loop;
658     int i;
659     struct in_addr addr;
660     struct ip_mreq *mreq;
661     struct ifnet *ifp;
662     struct ip_moptions *imo = *imop;
663     struct route ro;
664     struct sockaddr_in *dst;
665     if (imo == NULL) {
666         /*
667          * No multicast option buffer attached to the pcb;
668          * allocate one and initialize to default values.
669          */
670         imo = (struct ip_moptions *) malloc(sizeof(*imo), M_IPMOPTS,
671                                             M_WAITOK);
672         if (imo == NULL)
673             return (ENOBUFS);
674         *imop = imo;
675         imo->imo_multicast_ifp = NULL;
676         imo->imo_multicast_ttl = IP_DEFAULT_MULTICAST_TTL;
677         imo->imo_multicast_loop = IP_DEFAULT_MULTICAST_LOOP;
678         imo->imo_num_memberships = 0;
679     }
680     switch (optname) {
681         /* switch cases */
682     default:
683         error = EOPNOTSUPP;
684         break;
685     }
686     /*
687      * If all options have default values, no need to keep the structure.
688      */
689     if (imo->imo_multicast_ifp == NULL &&
690         imo->imo_multicast_ttl == IP_DEFAULT_MULTICAST_TTL &&
691         imo->imo_multicast_loop == IP_DEFAULT_MULTICAST_LOOP &&
692         imo->imo_num_memberships == 0) {
693         free(*imop, M_IPMOPTS);
694         *imop = NULL;
695     }
696     return (error);
697 }

```

---

ip\_output.c

## 650-664

The first argument, optname, indicates which multicast option is being changed. The second argument, imop, references a pointer to an ip\_moptions structure. If

\*imop is nonnull, ip\_setmoptions modifies the structure it points to. Otherwise, ip\_setmoptions allocates a new ip\_moptions structure and saves its address in \*imop. If no memory is available, ip\_setmoptions returns ENOBUFS immediately. Any subsequent errors that occur are posted in error, which is returned to the caller at the end of the function. The third argument, m, points to an mbuf that contains the data for the option to be changed (second column of Figure 12.16).

## Construct the defaults

665-679

When a new ip\_moptions structure is allocated, ip\_setmoptions initializes the default multicast interface pointer to null, initializes the default TTL to 1 (IP\_DEFAULT\_MULTICAST\_TTL), enables the loopback of multicast datagrams, and clears the group membership list. With these defaults, ip\_output selects an outgoing interface by consulting the

routing tables, multicasts are kept on the local network, and the system receives its own multicast transmissions if the outgoing interface is a member of the destination group.

## Process options

680-860

The body of ip\_setmoptions consists of a switch statement with a case for each option. The default case (for unknown options) sets error to EOPNOTSUPP.

## Discard structure if defaults are OK

861-872

After the switch statement, ip\_setmoptions examines the ip\_moptions structure. If all the multicast options match their respective default values, the structure is unnecessary and is released. ip\_setmoptions returns 0 or the posted error code.

## Selecting an Explicit Multicast Interface: **IP\_MULTICAST\_IF**

When optname is IP\_MULTICAST\_IF, the mbuf passed to ip\_setmoptions contains the unicast address of a multicast interface, which specifies the particular interface for multicasts sent on this socket. [Figure 12.20](#) shows the code for this option.

**Figure 12.20. ip\_setmoptions function:  
selecting a multicast output interface.**

---

```

681     case IP_MULTICAST_IF:
682     /*
683      * Select the interface for outgoing multicast packets.
684      */
685     if (m == NULL || m->m_len != sizeof(struct in_addr)) {
686         error = EINVAL;
687         break;
688     }
689     addr = *(mtod(m, struct in_addr *));
690     /*
691      * INADDR_ANY is used to remove a previous selection.
692      * When no interface is selected, a default one is
693      * chosen every time a multicast packet is sent.
694      */
695     if (addr.s_addr == INADDR_ANY) {
696         imo->imo_multicast_ifp = NULL;
697         break;
698     }
699     /*
700      * The selected interface is identified by its local
701      * IP address. Find the interface and confirm that
702      * it supports multicasting.
703      */
704     INADDR_TO_IFP(addr, ifp);
705     if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {
706         error = EADDRNOTAVAIL;
707         break;
708     }
709     imo->imo_multicast_ifp = ifp;
710     break;

```

---

ip\_output.c

## Validation

### 681-698

If no mbuf has been provided or the data within the mbuf is not the size of an `in_addr` structure, `ip_setmoptions` posts an `EINVAL` error; otherwise the data is copied into `addr`. If the interface address is `INADDR_ANY`, any previously selected interface is discarded. Subsequent multicasts with this `ip_moptions` structure are routed according to their destination group instead of through an explicitly

named interface ([Figure 12.40](#)).

## Select the default interface

699-710

If addr contains an address, INADDR\_TO\_IFP locates the matching interface. If a match can't be found or the interface does not support multicasting, EADDRNOTAVAIL is posted. Otherwise, ifp, the matching interface, becomes the multicast interface for output requests associated with this ip\_moptions structure.

## Selecting an Explicit Multicast TTL: IP\_MULTICAST\_TTL

When optname is IP\_MULTICAST\_TTL, the mbuf is expected to contain a single byte specifying the IP TTL for outgoing multicasts. This TTL is inserted by ip\_output into every multicast datagram sent on the associated socket. [Figure 12.21](#) shows the code for this option.

## Figure 12.21. ip\_setmoptions function: selecting an explicit multicast TTL.

```
711     case IP_MULTICAST_TTL:
712     /*
713      * Set the IP time-to-live for outgoing multicast packets.
714      */
715     if (m == NULL || m->m_len != 1) {
716         error = EINVAL;
717         break;
718     }
719     imo->imo_multicast_ttl = *(intod(m, u_char *));
720     break;
```

ip\_output.c

ip\_output.c

## Validate and select the default TTL

711-720

If the mbuf contains a single byte of data, it is copied into imo\_multicast\_ttl. Otherwise, EINVAL is posted.

## Selecting Multicast Loopbacks: **IP\_MULTICAST\_LOOP**

In general, multicast applications come in two forms:

- An application with one sender per system and multiple remote receivers. In this configuration only one local

process is sending datagrams to the group so there is no need to loopback outgoing multicasts. Examples include a multicast routing daemon and conferencing systems.

- An application with multiple senders and receivers on a system. Datagrams must be looped back so that each process receives the transmissions of the other senders on the system.

The IP\_MULTICAST\_LOOP option ([Figure 12.22](#)) selects the loopback policy associated with an ip\_moptions structure.

## Figure 12.22. ip\_setmoptions function: selecting multicast loopbacks.

```
721     case IP_MULTICAST_LOOP:
722         /*
723          * Set the loopback flag for outgoing multicast packets.
724          * Must be zero or one.
725         */
726         if (m == NULL || m->m_len != 1 ||
727             (loop = *(mtod(m, u_char *))) > 1) {
728             error = EINVAL;
729             break;
730         }
731         imo->imo_multicast_loop = loop;
732         break;
```

**Validate and select the loopback policy**

721-732

If  $m$  is null, does not contain 1 byte of data, or the byte is not 0 or 1, `EINVAL` is posted. Otherwise, the byte is copied into `imo_multicast_loop`. A 0 indicates that datagrams should not be looped back, and a 1 enables the loopback mechanism.

[Figure 12.23](#) shows the relationship between, the maximum scope of a multicast datagram, `imo_multicast_ttl`, and `imo_multicast_loop`.

### Figure 12.23. Loopback and TTL effects on multicast scope.

imo_multicast-		Recipients			
		Outgoing Interface?	Local Network?	Remote Networks?	Other Interfaces?
_loop	_ttl				
1	0	•			
1	1	•	•		
1	>1	•	•	•	see text

[Figure 12.23](#) shows that the set of interfaces that may receive a multicast packet depends on what the loopback policy is for the transmission and what TTL value is specified in the packet. A packet

may be received on an interface if the hardware receives its own transmissions, regardless of the loopback policy. A datagram may be routed through the network and arrive on another interface attached to the system ([Exercise 12.6](#)). If the sending system is itself a multicast router, outgoing packets may be forwarded to the other interfaces, but they will only be accepted for input processing on one interface ([Chapter 14](#)).

---



---

## Chapter 12. IP Multicasting

---

### 12.11 Joining an IP Multicast Group

Other than the IP all-hosts group, which the kernel maintains, joining or leaving a group is driven by explicit requests from processes. Joining (or leaving) a multicast group is more involved than joining (or leaving) an all-hosts group because the interface must be modified as well as any link-layer broadcast that we described for Ethernet.

The data passed in the mbuf when optname is `IP_MREQ` is shown in [Figure 12.24](#).

**Figure 12.24**

---

```
148 struct ip_mreq {
149     struct in_addr imr_multiaddr; /* IP multicast address of group */
150     struct in_addr imr_interface; /* local IP address of interface */
151 };
```

---

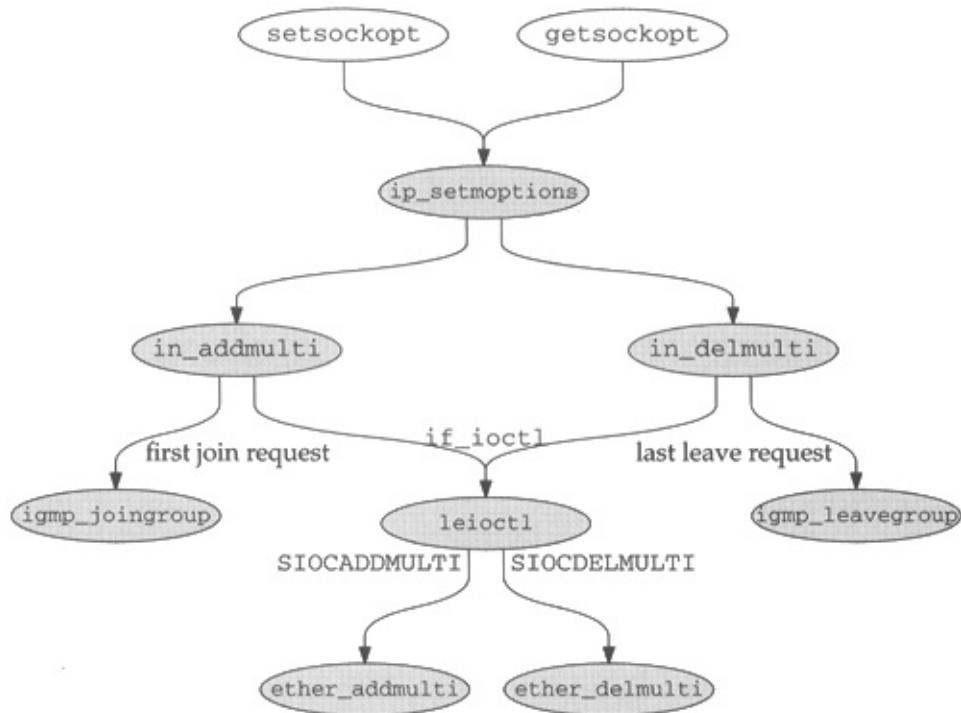
*in.h*

---

imr\_multiaddr specifies the multicast group and associated unicast IP address. The ip\_mreq structure specifies the interface index and the membership changes.

[Figure 12.25](#) illustrates the functions involved with our example Ethernet interface.

**Figure 12.25. Joining a Multicast Group**



We start by describing the changes to the `ip_mreq` structure in `ip_setmoptions` ([Figure 12.26](#)). Then we follow the flow of control through the driver, and to the physical device in our case, the Intel PRO/100 MT driver.

## **Figure 12.26. ip\_setmoptions**

```
733     case IP_ADD_MEMBERSHIP:  
734         /*  
735          * Add a multicast group membership.  
736          * Group must be a valid IP multicast address.  
737          */  
738         if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {  
739             error = EINVAL;  
740             break;  
741         }  
742         mreq = mtod(m, struct ip_mreq *);  
743         if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {  
744             error = EINVAL;  
745             break;  
746         }  
747         /*  
748          * If no interface address was provided, use the interface of  
749          * the route to the given multicast address.  
750          */  
751         if (mreq->imr_interface.s_addr == INADDR_ANY) {  
752             ro.ro_rt = NULL;  
753             dst = (struct sockaddr_in *) &ro.ro_dst;  
754             dst->sin_len = sizeof(*dst);  
755             dst->sin_family = AF_INET;  
756             dst->sin_addr = mreq->imr_multiaddr;  
757             rtaalloc(&ro);  
758             if (ro.ro_rt == NULL) {  
759                 error = EADDRNOTAVAIL;  
760                 break;  
761             }  
762             ifp = ro.ro_rt->rt_ifp;  
763             rtfree(ro.ro_rt);  
764         } else {  
765             INADDR_TO_IFP(mreq->imr_interface, ifp);  
766         }  
767         /*  
768          * See if we found an interface, and confirm that it  
769          * supports multicast.  
770          */  
771         if (ifp == NULL || (ifp->if_flags & IFF_MULTICAST) == 0) {  
772             error = EADDRNOTAVAIL;  
773             break;  
774         }  
775         /*  
776          * See if the membership already exists or if all the  
777          * membership slots are full.  
778          */  
779         for (i = 0; i < imo->imo_num_memberships; ++i) {  
780             if (imo->imo_membership[i]->inm_ifp == ifp &&  
781                 imo->imo_membership[i]->inm_addr.s_addr  
782                 == mreq->imr_multiaddr.s_addr)  
783                 break;  
784         }  
785         if (i < imo->imo_num_memberships) {  
786             error = EADDRINUSE;  
787             break;  
788         }  
789         if (i == IP_MAX_MEMBERSHIPS) {  
790             error = ETOOMANYREFS;  
791             break;  
792         }  
793         /*  
794          * Everything looks good; add a new record to the multicast  
795          * address list for the given interface.  
796          */  
797         if ((imo->imo_membership[i] =  
798             in_addmulti(&mreq->imr_multiaddr, ifp)) == NULL) {  
799             error = ENOBUFS;  
800             break;  
801         }  
802         ++imo->imo_num_memberships;  
803         break;  
804     }  
805     ip_output.c
```

## Validation

733-746

ip\_setmoptions starts by validating the request if the address (imr\_multiaddr) within the struct posts EINVAL. mreq points to the valid ip\_mrec

## Locate the interface

747-774

If the unicast address of the interface (imr\_interface) is INADDR\_ANY, it becomes the default interface for the specified group. A pointer to the desired destination and passed to rtalloc, which returns a route. If the add request fails with the error EADDRNOTAVAIL, the interface for the route is saved in ifp and the rc is set to -1.

If imr\_interface is not INADDR\_ANY, an explicit search is performed using INADDR\_TO\_IFP. It searches for the interface with the specified address. If no interface is found or if it does not support multicasting, the function returns -1.

We described the route structure in [Section 8](#), and the use of the routing tables for selecting

## Already a member?

775-792

The last check performed on the request is to ensure the selected interface is already a member of the requested group. If the membership array is full, EADDRINUSE or ETOOMANYMEMBERS errors stops.

## Join the group

793-803

At this point the request looks reasonable. `in_addrlist` contains the datagrams for the group. The pointer returned by `in_ifmultiinfo` is updated to point to the new structure ([Figure 12.12](#)) in the interface's multiarray. The size of the array is incremented.

## in\_addmulti Function

`in_addmulti` and its companion `in_delmulti` ([Figure 12.26](#)) handle the groups that an interface has joined. Join requests add an interface to the list or increase the reference count of an existing one.

**Figure 12.27. `in_addmulti` Function**

```
469 struct in_multi *
470 in_addmulti(ap, ifp)
471 struct in_addr *ap;
472 struct ifnet *ifp;
473 {
474     struct in_multi *inm;
475     struct ifreq ifr;
476     struct in_ifaddr *ia;
477     int     s = splnet();
478     /*
479      * See if address already in list.
480      */
481     IN_LOOKUP_MULTI(*ap, ifp, inm);
482     if (inm != NULL) {
483         /*
484          * Found it; just increment the reference count.
485          */
486         ++inm->inm_refcount;
487     } else {
```

in.c

## Already a member

469-487

ip\_setmoptions has already verified that ap points to a multicast-capable interface. IN\_LOOKUP\_MULTI already a member of the group. If it is a member, it returns.

If the interface is not yet a member of the group,

Figure 12.28. in\_add

---

```

487     ) else {
488         /*
489          * New address; allocate a new multicast record
490          * and link it into the interface's multicast list.
491          */
492         inm = (struct in_multi *) malloc(sizeof(*inm),
493                                         M_IPMADDR, M_NOWAIT);
494         if (inm == NULL) {
495             splx(s);
496             return (NULL);
497         }
498         inm->inm_addr = *ap;
499         inm->inm_ifp = ifp;
500         inm->inm_refcount = 1;
501         IPP_TO_IA(ifp, ia);
502         if (ia == NULL) {
503             free(inm, M_IPMADDR);
504             splx(s);
505             return (NULL);
506         }
507         inm->inm_ia = ia;
508         inm->inm_next = ia->ia_multiaddrs;
509         ia->ia_multiaddrs = inm;
510         /*
511          * Ask the network driver to update its multicast reception
512          * filter appropriately for the new address.
513          */
514         ((struct sockaddr_in *)&ifr.ifr_addr)->sin_family = AF_INET;
515         ((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr = *ap;
516         if ((ifp->if_ioctl == NULL) ||
517             (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) & ifr) != 0) {
518             ia->ia_multiaddrs = inm->inm_next;
519             free(inm, M_IPMADDR);
520             splx(s);
521             return (NULL);
522         }
523         /*
524          * Let IGMP know that we have joined a new IP multicast group.
525          */
526         igmp_joingroup(inm);
527     }
528     splx(s);
529     return (inm);
530 }

```

---

*in.c*

## Update the `in_multi` list

487-509

If the interface isn't a member yet, `in_addmult` structure at the front of the `ia_multiaddrs` list is

Update the interface and announce the change

## 510-530

If the interface driver has defined an if\_ioctl function (Figure 4.23) containing the group address and If the interface rejects the request, the in\_mult function returns -1. Finally, in\_addmulti calls igmp\_joingroup to prc routers.

in\_addmulti returns a pointer to the in\_multi structure.

## sioctl and Iioctl Functions: SIOCADDMULTI

Multicast group processing for the SLIP and loopback drivers involves more than error checking. Figure 12.29 shows the SLIP implementation of the SIOCADDMULTI ioctl.

Figure 12.29. sioctl function for SLIP

```
673     case SIOCADDMULTI:
674     case SIOCDELMULTI:
675         ifr = (struct ifreq *) data;
676         if (ifr == 0) {
677             error = EAFNOSUPPORT; /* XXX */
678             break;
679         }
680         switch (ifr->ifr_addr.sa_family) {
681             case AF_INET:
682                 break;
683             default:
684                 error = EAFNOSUPPORT;
685                 break;
686         }
687     break;
```

## 673-687

EAFNOSUPPORT is returned whether the request is supported.

Figure 12.30 shows the loopback processing.

### Figure 12.30. ioctl function for loopback processing

```
if_loop.c
152     case SIOCADDMULTI:
153     case SIOCDELMULTI:
154         ifr = (struct ifreq *) data;
155         if (ifr == 0) {
156             error = EAFNOSUPPORT; /* XXX */
157             break;
158         }
159         switch (ifr->ifr_addr.sa_family) {
160             case AF_INET:
161                 break;
162             default:
163                 error = EAFNOSUPPORT;
164                 break;
165         }
166     break;
```

if\_loop.c

152-166

The processing for the loopback interface is identical to Figure 12.29. EAFNOSUPPORT is returned whether the request is supported.

### ioctl Function: SIOCADDMULTI and SIOCDELMULTI

Recall from Figure 4.2 that ioctl is the if\_ioctl function. Figure 12.31 shows the code for the SIOCADDMULTI and SIOCDELMULTI requests.

### Figure 12.31. ioctl function for SIOCADDMULTI and SIOCDELMULTI

```

657     case SIOCADDMULTI:
658     case SIOCDELMULTI:
659         /* Update our multicast list */
660         error = (cmd == SIOCADDMULTI) ?
661             ether_addmulti((struct ifreq *) data, &le->sc_ac) :
662             ether_delmulti((struct ifreq *) data, &le->sc_ac);
663
664         if (error == ENETRESET) {
665             /*
666              * Multicast list has changed; set the hardware
667              * filter accordingly.
668              */
669             lereset(ifp->if_unit);
670             error = 0;
671         }
672     break;

```

if\_le.c

## 657-671

Ieioctl passes add and delete requests directly to the hardware. Both functions return ENETRESET if the request cannot be received by the physical hardware. If this occurs, it updates the new multicast reception list.

We don't show lereset, as it is specific to the Intel driver. It arranges for the hardware to receive frames whose addresses are contained in the ether\_multi list associated with the interface. If each entry on the multicast list specifies a hardware mechanism to receive multicast packets selectively, or a range of addresses, it abandons the hash strategy and receives all multicast packets. If the driver must fall back to software processing, the IFF\_ALLMULTI flag is on when lereset returns.

## ether\_addmulti Function

Every Ethernet driver calls ether\_addmulti to program the hardware to map the IP class D address to the appropriate hardware multicast address.

the ether\_multi list. Figure 12.32 shows the first part of the code for ether\_addmulti.

Figure 12.32. ether\_addmulti

```
if_ether.c
366 int
367 ether_addmulti(ifr, ac)
368 struct ifreq *ifr;
369 struct arpcom *ac;
370 {
371     struct ether_multi *enm;
372     struct sockaddr_in *sin;
373     u_char    addrlo[6];
374     u_char    addrhi[6];
375     int      s = splimp();
376
377     switch (ifr->ifr_addr.sa_family) {
378
379         case AF_UNSPEC:
380             bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
381             bcopy(addrlo, addrhi, 6);
382             break;
383
384         case AF_INET:
385             sin = (struct sockaddr_in *) &(ifr->ifr_addr);
386             if (sin->sin_addr.s_addr == INADDR_ANY) {
387                 /*
388                  * An IP address of INADDR_ANY means listen to all
389                  * of the Ethernet multicast addresses used for IP.
390                  * (This is for the sake of IP multicast routers.)
391                 */
392                 bcopy(ether_ipmulticast_min, addrlo, 6);
393                 bcopy(ether_ipmulticast_max, addrhi, 6);
394             } else {
395                 ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
396                 bcopy(addrlo, addrhi, 6);
397             }
398             break;
399
400     default:
401         splx(s);
402         return (EAFNOSUPPORT);
403     }
}
```

if\_ether.c

## Initialize address range

366-399

First, ether\_addmulti initializes a range of mult

of six unsigned characters). If the requested address is an explicit Ethernet multicast address, the address is in the AF\_INET family and is INADDR\_BROADCAST to ether\_ipmulticast\_min and addrhi to ether\_ipmulticast\_max. These addresses are defined as:

```
u_char ether_ipmulticast_min[6]
u_char ether_ipmulticast_max[6]
```

As with etherbroadcastaddr ([Section 4.3](#)), this

IP multicast routers must listen for all IP multicast addresses. An IP multicast router is considered a request to join every IP multicast group in its broadcast domain. This case spans the entire block of IP multicast addresses.

The mrouted(8) daemon issues a SIOCADDMLROUTE command to add routing packets for a multicast interface.

ETHER\_MAP\_IP\_MULTICAST maps any other specific IP multicast address. Requests for other address formats are mapped to the broadcast address.

While the Ethernet multicast list supports address ranges, it is used to request a specific range, other than to enumerate them. The range always set to the same address.

The second half of ether\_addmulti, shown in [Figure 4.10](#), adds the address to the list if it is new.

## Figure 12.33. ether\_ad

```
400  /*
401   * Verify that we have valid Ethernet multicast addresses.
402   */
403  if ((addrlo[0] & 0x01) != 1 || (addrhi[0] & 0x01) != 1) {
404      splx(s);
405      return (EINVAL);
406  }
407  /*
408   * See if the address range is already in the list.
409   */
410  ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
411  if (enm != NULL) {
412      /*
413       * Found it; just increment the reference count.
414       */
415      ++enm->enm_refcount;
416      splx(s);
417      return (0);
418  }
419  /*
420   * New address or range; malloc a new multicast record
421   * and link it into the interface's multicast list.
422   */
423  enm = (struct ether_multi *) malloc(sizeof(*enm), M_IFMADDR, M_NOWAIT);
424  if (enm == NULL) {
425      splx(s);
426      return (ENOBUFS);
427  }
428  bcopy(addrlo, enm->enm_addrlo, 6);
429  bcopy(addrhi, enm->enm_addrhi, 6);
430  enm->enm_ac = ac;
431  enm->enm_refcount = 1;
432  enm->enm_next = ac->ac_multiaddrs;
433  ac->ac_multiaddrs = enm;
434  ac->ac_multicnt++;
435  splx(s);
436  /*
437   * Return ENETRESET to inform the driver that the list has changed
438   * and its reception filter should be adjusted accordingly.
439   */
440  return (ENETRESET);
441 }
```

if\_ETHERSUBR.C

## Already receiving

400-418

ether\_addmulti checks the multicast bit ([Figure](#) they are indeed Ethernet multicast addresses. I

hardware is already listening for the specified range (enm\_refcount) in the matching ether\_multi structure.

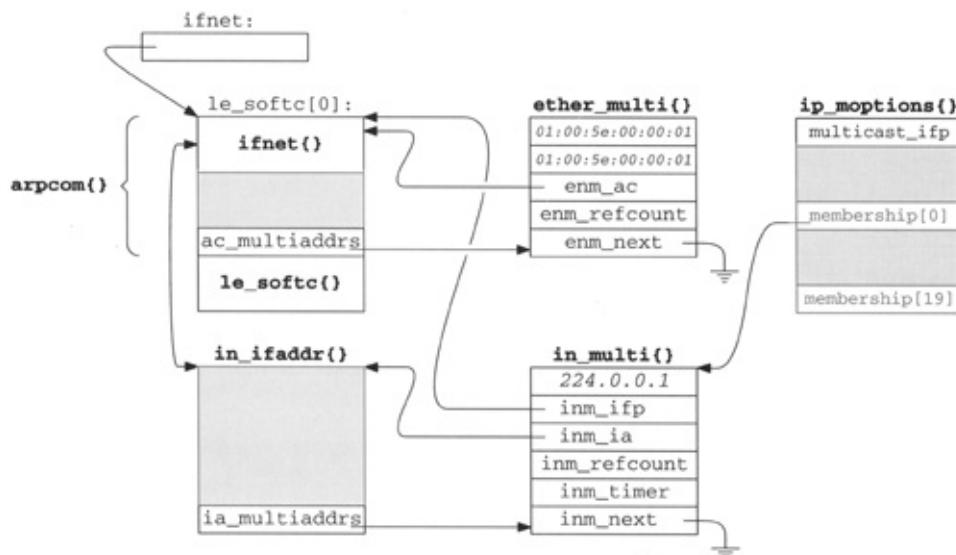
## Update ether\_multi list

419-441

If this is a new address range, a new ether\_multi structure ac\_multiaddrs list in the interface's arpcom structure ether\_addmulti, the device driver that called the function, and the hardware reception filter must be updated.

Figure 12.34 shows the relationships between the structures after the LANCE Ethernet interface has joined the network.

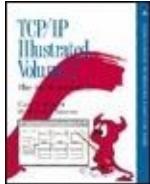
Figure 12.34. Overview of the ether\_multi list



---

**Team-Fly**





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.12 Leaving an IP Multicast Group

In general, the steps required to leave a group are the reverse of those required to join a group. The membership list in the ip\_moptions structure is updated, the in\_multi list for the IP interface is updated, and the ether\_multi list for the device is updated. First, we return to ip\_setmoptions and the IP\_DROP\_MEMBERSHIP case, which we show in Figure 12.35.

**Figure 12.35. ip\_setmoptions function:  
leaving a multicast group.**

```

804     case IP_DROP_MEMBERSHIP:
805     /*
806      * Drop a multicast group membership.
807      * Group must be a valid IP multicast address.
808      */
809     if (m == NULL || m->m_len != sizeof(struct ip_mreq)) {
810         error = EINVAL;
811         break;
812     }
813     mreq = mtod(m, struct ip_mreq *);
814     if (!IN_MULTICAST(ntohl(mreq->imr_multiaddr.s_addr))) {
815         error = EINVAL;
816         break;
817     }
818     /*
819      * If an interface address was specified, get a pointer
820      * to its ifnet structure.
821      */
822     if (mreq->imr_interface.s_addr == INADDR_ANY)
823         ifp = NULL;
824     else {
825         INADDR_TO_IFP(mreq->imr_interface, ifp);
826         if (ifp == NULL) {
827             error = EADDRNOTAVAIL;
828             break;
829         }
830     }
831     /*
832      * Find the membership in the membership array.
833      */
834     for (i = 0; i < imo->imo_num_memberships; ++i) {
835         if ((ifp == NULL ||
836             imo->imo_membership[i]->inm_ifp == ifp) &&
837             imo->imo_membership[i]->inm_addr.s_addr ==
838             mreq->imr_multiaddr.s_addr)
839             break;
840     }
841     if (i == imo->imo_num_memberships) {
842         error = EADDRNOTAVAIL;
843         break;
844     }
845     /*
846      * Give up the multicast address record to which the
847      * membership points.
848      */
849     in_delmulti(imo->imo_membership[i]);
850     /*
851      * Remove the gap in the membership array.
852      */
853     for (++i; i < imo->imo_num_memberships; ++i)
854         imo->imo_membership[i - 1] = imo->imo_membership[i];
855     --imo->imo_num_memberships;
856     break;

```

---

ip\_output.c

## Validation

804-830

The mbuf must contain an ip\_mreq structure, within the structure

`imr_multiaddr` must be a multicast group, and there must be an interface associated with the unicast address `imr_interface`. If these conditions aren't met, `EINVAL` or `EADDRNOTAVAIL` is posted and processing continues at the end of the switch.

## Delete membership references

831-856

The for loop searches the group membership list for an `in_multi` structure with the requested {interface, group} pair. If a match isn't found, `EADDRNOTAVAIL` is posted. Otherwise, `in_delmulti` updates the `in_multi` list and the second for loop removes the unused entry in the membership array by shifting subsequent entries to fill the gap. The size of the array is updated accordingly.

## `in_delmulti` Function

Since many processes may be receiving multicast datagrams, calling `in_delmulti` ([Figure 12.36](#)) results only in leaving the

specified group when there are no more references to the in\_multi structure.

## Figure 12.36. in\_delmulti function.

```
in.c
534 int
535 in_delmulti(inm)
536 struct in_multi *inm;
537 {
538     struct in_multi **p;
539     struct ifreq ifr;
540     int     s = splnet();
541
542     if (--inm->inm_refcount == 0) {
543         /*
544          * No remaining claims to this record; let IGMP know that
545          * we are leaving the multicast group.
546        */
547         igmp_leavegroup(inm);
548         /*
549          * Unlink from list.
550        */
551         for (p = &inm->inm_ia->ia_multiaddrs;
552             *p != inm;
553             p = &(*p)->inm_next)
554             continue;
555         *p = (*p)->inm_next;
556         /*
557          * Notify the network driver to update its multicast reception
558          * filter.
559        */
560         ((struct sockaddr_in *)&(ifr.ifr_addr))->sin_family = AF_INET;
561         ((struct sockaddr_in *)&(ifr.ifr_addr))->sin_addr =
562             inm->inm_addr;
563         (*inm->inm_ifp->if_ioctl) (inm->inm_ifp, SIOCDELMULTI,
564                                     (caddr_t) & ifr);
565         free(inm, M_IPMADDR);
566     }
567     splx(s);
568 }
```

## Update in\_multi structure

534-567

in\_delmulti starts by decrementing the reference count of the in\_multi structure

and returning if the reference count is nonzero. If the reference count drops to 0, there are no longer any processes waiting for the multicast datagrams on the specified {interface, group} pair. `igmp_leavegroup` is called, but as we'll see in [Section 13.8](#), the function does nothing.

The for loop traverses the linked list of `in_multi` structures until it locates the matching structure.

The body of this for loop consists of the single `continue` statement. All the work is done by the expressions at the top of the loop. The `continue` is not required but stands out more clearly than a bare semicolon.

The `ETHER_LOOKUP_MULTI` macro in [Figure 12.9](#) does not use the `continue` and the bare semicolon is almost undetectable.

After the loop, the matching `in_multi` structure is unlinked and `in_delmulti` issues the `SIOCDELMULTI` request to the interface so that any device-specific data

structures can be updated. For Ethernet interfaces, this means the ether\_multi list is updated. Finally, the in\_multi structure is released.

The SIOCDELMULTI case for the LANCE driver was included in [Figure 12.31](#) where we also discussed the SIOCADDMULTI case.

## **ether\_delmulti Function**

When IP releases an in\_multi structure associated with an Ethernet device, the device may be able to release the matching ether\_multi structure. We say *may* because IP may be unaware of other software listening for IP multicasts. When the reference count for the ether\_multi structure drops to 0, it can be released. [Figure 12.37](#) shows the ether\_delmulti function.

**Figure 12.37. ether\_delmulti function.**

```
445 int
446 ether_delmulti(ifr, ac)
447 struct ifreq *ifr;
448 struct arpcom *ac;
449 {
450     struct ether_multi *enm;
451     struct ether_multi **p;
452     struct sockaddr_in *sin;
453     u_char    addrlo[6];
454     u_char    addrhi[6];
455     int      s = splimp();
456
457     switch (ifr->ifr_addr.sa_family) {
458
459         case AF_UNSPEC:
460             bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
461             bcopy(addrlo, addrhi, 6);
462             break;
463
464         case AF_INET:
465             sin = (struct sockaddr_in *) &(ifr->ifr_addr);
466             if (sin->sin_addr.s_addr == INADDR_ANY) {
467                 /*
468                  * An IP address of INADDR_ANY means stop listening
469                  * to the range of Ethernet multicast addresses used
470                  * for IP.
471                 */
472                 bcopy(ether_ipmulticast_min, addrlo, 6);
473                 bcopy(ether_ipmulticast_max, addrhi, 6);
474             } else {
475                 ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
476                 bcopy(addrlo, addrhi, 6);
477             }
478             break;
479
480         default:
481             splx(s);
482             return (EAFNOSUPPORT);
483     }
484
485     /*
486      * Look up the address in our list.
487      */
488     ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
489     if (enm == NULL) {
490         splx(s);
491         return (ENXIO);
492     }
493     if (--enm->enm_refcount != 0) {
494         /*
495          * Still some claims to this record.
496          */
497         splx(s);
498         return (0);
499     }
```

```
495     /*
496      * No remaining claims to this record; unlink and free it.
497      */
498     for (p = &enm->enm_ac->ac_multiaddrs;
499          *p != enm;
500          p = &(*p)->enm_next)
501         continue;
502     *p = (*p)->enm_next;
503     free(enm, M_IFMADDR);
504     ac->ac_multicnt--;
505     splx(s);
506     /*
507      * Return ENETRESET to inform the driver that the list has changed
508      * and its reception filter should be adjusted accordingly.
509      */
510     return (ENETRESET);
511 }
```

if\_ether.c

## 445-479

ether\_delmulti initializes the addrlo and addrhi arrays in the same way as ether\_addmulti does.

## Locate ether\_multi structure

## 480-494

ETHER\_LOOKUP\_MULTI locates a matching ether\_multi structure. If it isn't found, ENXIO is returned. If the matching structure is found, the reference count is decremented and if the result is nonzero, ether\_delmulti returns immediately. In this case, the structure may not be released because another protocol has elected to receive the same multicast packets.

## **Delete ether\_multi structure**

**495-511**

The for loop searches the ether\_multi list for the matching address range. The matching structure is unlinked from the list and released. Finally, the size of the list is updated and ENETRESET is returned so that the device driver can update its hardware reception filter.

---

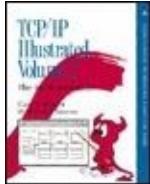
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.13 ip\_getmoptions Function

Fetching the current option settings is considerably easier than setting them. All the work is done by `ip_getmoptions`, shown in [Figure 12.38](#).

**Figure 12.38. `ip_getmoptions` function.**

---

```

876 int ip_getmoptions(optname, imo, mp)
877 int optname;
878 struct ip_moptions *imo;
879 struct mbuf **mp;
880 struct in_ifaddr *ia;
881 {
882     u_char *ttl;
883     u_char *loop;
884     struct in_addr *addr;
885     struct in_ifaddr *ia;
886     *mp = m_get(M_WAIT, MT_SOOPTS);
887     switch (optname) {
888     case IP_MULTICAST_IF:
889         addr = mtod(*mp, struct in_addr *);
890         (*mp)->m_len = sizeof(struct in_addr);
891         if (imo == NULL || imo->imo_multicast_ifp == NULL)
892             addr->s_addr = INADDR_ANY;
893         else {
894             IFF_TO_IA(imo->imo_multicast_ifp, ia);
895             addr->s_addr = (ia == NULL) ? INADDR_ANY
896                                         : IA_SIN(ia)->sin_addr.s_addr;
897         }
898         return (0);
899     case IP_MULTICAST_TTL:
900         ttl = mtod(*mp, u_char *);
901         (*mp)->m_len = 1;
902         *ttl = (imo == NULL) ? IP_DEFAULT_MULTICAST_TTL
903                         : imo->imo_multicast_ttl;
904         return (0);
905     case IP_MULTICAST_LOOP:
906         loop = mtod(*mp, u_char *);
907         (*mp)->m_len = 1;
908         *loop = (imo == NULL) ? IP_DEFAULT_MULTICAST_LOOP
909                 : imo->imo_multicast_loop;
910         return (0);
911     default:
912         return (EOPNOTSUPP);
913     }
914 }
```

---

ip\_output.c

## Copy the option data and return

876-914

The three arguments to ip\_getmoptions are: optname, the option to fetch; imo, the ip\_moptions structure; and mp, which points to a pointer to an mbuf. m\_get

allocates an mbuf to hold the option data. For each of the three options, a pointer (addr, ttl, and loop, respectively) is initialized to the data area of the mbuf and the length of the mbuf is set to the length of the option data.

For IP\_MULTICAST\_IF, the unicast address found by IFP\_TO\_IA is returned or INADDR\_ANY is returned if no explicit multicast interface has been selected.

For IP\_MULTICAST\_TTL, imo\_multicast\_ttl is returned or if an explicit multicast TTL has not been selected, 1 (IP\_DEFAULT\_MULTICAST\_TTL) is returned.

For IP\_MULTICAST\_LOOP, imo\_multicast\_loop is returned or if an explicit multicast loopback policy has not been selected, 1 (IP\_DEFAULT\_MULTICAST\_LOOP) is returned.

Finally, EOPNOTSUPP is returned if the option isn't recognized.

---

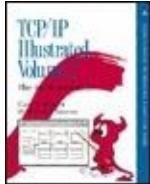
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.14 Multicast Input Processing: ipintr Function

Now that we have described multicast addressing, group memberships, and the various data structures associated with IP and Ethernet multicasting, we can move on to multicast datagram processing.

In [Figure 4.13](#) we saw that an incoming Ethernet multicast packet is detected by ether\_input, which sets the M\_MCAST flag in the mbuf header before placing an IP packet on the IP input queue (ipintrq). The ipintr function processes each packet in turn. The multicast processing code we omitted from the discussion of ipintr appears in [Figure 12.39](#).

## Figure 12.39. ipintr function: multicast input processing.

```
214     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {----- ip_input.c
215         struct in_multi *inm;
216         extern struct socket *ip_mrouter;
217
218         if (ip_mrouter) {
219             /*
220              * If we are acting as a multicast router, all
221              * incoming multicast packets are passed to the
222              * kernel-level multicast forwarding function.
223              * The packet is returned (relatively) intact; if
224              * ip_mforward() returns a non-zero value, the packet
225              * must be discarded, else it may be accepted below.
226              *
227              * (The IP ident field is put in the same byte order
228              * as expected when ip_mforward() is called from
229              * ip_output().)
230             */
231             ip->ip_id = htons(ip->ip_id);
232             if (ip_mforward(m, m->m_pkthdr.recvif) != 0) {
233                 ipstat.ips_cantforward++;
234                 m_freem(m);
235                 goto next;
236             }
237             ip->ip_id = ntohs(ip->ip_id);
238
239             /*
240              * The process-level routing demon needs to receive
241              * all multicast IGMP packets, whether or not this
242              * host belongs to their destination groups.
243              */
244             if (ip->ip_p == IPPROTO_IGMP)
245                 goto ours;
246             ipstat.ips_forward++;
247
248             /*
249              * See if we belong to the destination multicast group on the
250              * arrival interface.
251              */
252             IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.recvif, inm);
253             if (inm == NULL) {
254                 ipstat.ips_cantforward++;
255                 m_freem(m);
256                 goto next;
257             }
258             goto ours;
259     }
```

----- ip\_input.c

The code is from the section of ipintr that determines if a packet is addressed to the local system or if it should be forwarded. At this point, the packet has been checked

for errors and any options have been processed. ip points to the IP header within the packet.

## Forward packets if configured as multicast router

214-245

This entire section of code is skipped if the destination address is not an IP multicast group. If the address is a multicast group and the system is configured as an IP multicast router (ip\_mrouter), ip\_id is converted to network byte order (the form that ip\_mforward expects), and the packet is passed to ip\_mforward. If ip\_mforward returns a nonzero value, an error was detected or the packet arrived through a *multicast tunnel*. The packet is discarded and ips\_cantforward incremented.

We describe multicast tunnels in [Chapter 14](#). They transport multicast packets between multicast routers separated by standard IP routers. Packets that arrive through a tunnel

must be processed by ip\_mforward and not ipintr.

If ip\_mforward returns 0, ip\_id is converted back to host byte order and ipintr may continue processing the packet.

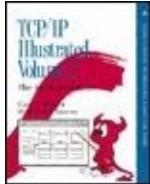
If ip points to an IGMP packet, it is accepted and execution continues at ours (ipintr, [Figure 10.11](#)). A multicast router must accept all IGMP packets irrespective of their individual destination groups or of the group memberships of the incoming interface. The IGMP packets contain announcements of membership changes.

246-257

The remaining code in [Figure 12.39](#) is executed whether or not the system is configured as a multicast router. IN\_LOOKUP\_MULTI searches the list of multicast groups that the interface has joined. If a match is not found, the packet is discarded. This occurs when the hardware filter accepts unwanted packets or when a group associated with the interface and the destination group of the

packet map to the same Ethernet multicast address.

If the packet is accepted, execution continues at the label ours in ipintr ([Figure 10.11](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.15 Multicast Output Processing: ip\_output Function

When we discussed `ip_output` in [Chapter 8](#), we postponed discussion of the `mp` argument to `ip_output` and the multicast processing code. In `ip_output`, if `mp` points to an `ip_moptions` structure, it overrides the default multicast output processing. The omitted code from `ip_output` appears in [Figures 12.40](#) and [12.41](#). `ip` points to the outgoing packet, `m` points to the mbuf holding the packet, and `ifp` points to the interface selected by the routing tables for the destination group.

**Figure 12.40. `ip_output` function: defaults**

**and source address.**

```
129     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) { ip_output.c
130         struct in_multi *inm;
131         extern struct ifnet loif;
132
133         m->m_flags |= M_MCAST;
134         /*
135          * IP destination address is multicast.  Make sure "dst"
136          * still points to the address in *ro".  (It may have been
137          * changed to point to a gateway address, above.)
138          */
139         dst = (struct sockaddr_in *) &ro->ro_dst;
140         /*
141          * See if the caller provided any multicast options
142          */
143         if (imo != NULL) {
144             ip->ip_ttl = imo->imo_multicast_ttl;
145             if (imo->imo_multicast_ifp != NULL)
146                 ifp = imo->imo_multicast_ifp;
147             } else
148                 ip->ip_ttl = IP_DEFAULT_MULTICAST_TTL;
149             /*
150              * Confirm that the outgoing interface supports multicast.
151              */
152             if ((ifp->if_flags & IFF_MULTICAST) == 0) {
153                 ipstat.ips_noroute++;
154                 error = ENETUNREACH;
155                 goto bad;
156             }
157             /*
158              * If source address not specified yet, use address
159              * of outgoing interface.
160              */
161             if (ip->ip_src.s_addr == INADDR_ANY) {
162                 struct in_ifaddr *ia;
163
164                 for (ia = in_ifaddr; ia; ia = ia->ia_next)
165                     if (ia->ia_ifp == ifp) {
166                         ip->ip_src = IA_SIN(ia)->sin_addr;
167                         break;
168                     }
169             }
```

**Figure 12.41. ip\_output function: loopback, forward, and send.**

```

168     IN_LOOKUP_MULTI(ip->ip_dst, ifp, inm);
169     if (inm != NULL &&
170         (imo == NULL || imo->imo_multicast_loop)) {
171         /*
172          * If we belong to the destination multicast group
173          * on the outgoing interface, and the caller did not
174          * forbid loopback, loop back a copy.
175          */
176         ip_mloopback(ifp, m, dst);
177     } else {
178         /*
179          * If we are acting as a multicast router, perform
180          * multicast forwarding as if the packet had just
181          * arrived on the interface to which we are about
182          * to send. The multicast forwarding function
183          * recursively calls this function, using the
184          * IP_FORWARDING flag to prevent infinite recursion.
185          *
186          * Multicasts that are looped back by ip_mloopback(),
187          * above, will be forwarded by the ip_input() routine,
188          * if necessary.
189          */
190         extern struct socket *ip_mrouted;
191         if (ip_mrouted && (flags & IP_FORWARDING) == 0) {
192             if (ip_mforward(m, ifp) != 0) {
193                 m_freem(m);
194                 goto done;
195             }
196         }
197     }
198     /*
199      * Multicasts with a time-to-live of zero may be looped-
200      * back, above, but must not be transmitted on a network.
201      * Also, multicasts addressed to the loopback interface
202      * are not sent -- the above call to ip_mloopback() will
203      * loop back a copy if this host actually belongs to the
204      * destination group on the loopback interface.
205      */
206     if (ip->ip_ttl == 0 || ifp == &loif) {
207         m_freem(m);
208         goto done;
209     }
210     goto sendit;
211 }
```

ip\_output.c

## Establish defaults

### 129-155

The code in Figure 12.40 is executed only if the packet is destined for a multicast group. If so, ip\_output sets M\_MCAST in the mbuf and dst is reset to the final

destination as it may have been set to the next-hop router earlier in ip\_output (Figure 8.24).

If an ip\_moptions structure was passed, ip\_ttl and ifp are changed accordingly. Otherwise, ip\_ttl is set to 1 (IP\_DEFAULT\_MULTICAST\_TTL), which prevents the multicast from escaping to a remote network. The interface selected by consulting the routing tables or the interface specified within the ip\_moptions structure must support multicasting. If it does not, ip\_output discards the packet and returns ENETUNREACH.

## Select source address

156-167

If the source address is unspecified, the for loop finds the Internet unicast address associated with the outgoing interface and fills in ip\_src in the IP header.

Unlike a unicast packet, an outgoing multicast packet may be transmitted on

more than one interface if the system is configured as a multicast router. Even if the system is not a multicast router, the outgoing interface may be a member of the destination group and may need to receive the packet. Finally, we need to consider the multicast loopback policy and the loopback interface itself. Taking all this into account, there are three questions to consider:

- Should the packet be received on the outgoing interface?
- Should the packet be forwarded to other interfaces?
- Should the packet be transmitted on the outgoing interface?

[Figure 12.41](#) shows the code from ip\_output that answers these questions.

## Loopback or not?

168-176

If IN\_LOOKUP\_MULTI determines that the

outgoing interface is a member of the destination group and `imo_multicast_loop` is nonzero, the packet is queued for *input* on the output interface by `ip_mloopback`. In this case, the original packet is *not* considered for forwarding, since the copy is forwarded during input processing if necessary.

## Forward or not?

178-197

If the packet is *not* looped back, but the system is configured as a multicast router and the packet is eligible for forwarding, `ip_mforward` distributes copies to other multicast interfaces. If `ip_mforward` does not return 0, `ip_output` discards the packet and does not attempt to transmit it. This indicates an error with the packet.

To prevent infinite recursion between `ip_mforward` and `ip_output`, `ip_mforward` always turns on `IP_FORWARDING` before calling `ip_output`. A datagram originating on the system is eligible for forwarding

because the transport protocols do not turn on IP\_FORWARDING.

## Transmit or not?

198-209

Packets with a TTL of 0 may be looped back, but they are never forwarded (ip\_mforward discards them) and are never transmitted. If the TTL is 0 or if the output interface is the loopback interface, ip\_output discards the packet since the TTL has expired or the packet has already been looped back by ip\_mloopback.

## Send packet

210-211

If the packet has made it this far, it is ready to be physically transmitted on the output interface. The code at sendit (ip\_output, [Figure 8.25](#)) may fragment the datagram before passing it (or the resulting fragments) to the interface's if\_output function. We'll see in [Section](#)

[21.10](#) that the Ethernet output function, ether\_output, calls arpresolve, which calls ETHER\_MAP\_IP\_MULTICAST to construct an Ethernet multicast destination address based on the IP multicast destination address.

## ip\_mloopback Function

ip\_mloopback relies on looutput ([Figure 5.27](#)) to do its job. Instead of passing a pointer to the loopback interface to looutput, ip\_mloopback passes a pointer to the output multicast interface. The ip\_mloopback function is shown in [Figure 12.42](#).

**Figure 12.42. ip\_mloopback function.**

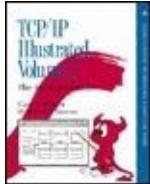
```
935 static void  
936 ip_mloopback(ifp, m, dst)  
937 struct ifnet *ifp;  
938 struct mbuf *m;  
939 struct sockaddr_in *dst;  
940 {  
941     struct ip *ip;  
942     struct mbuf *copym;  
943     copym = m_copy(m, 0, M_COPYALL);  
944     if (copym != NULL) {  
945         /*  
946          * We don't bother to fragment if the IP length is greater  
947          * than the interface's MTU.  Can this possibly matter?  
948          */  
949         ip = mtod(copym, struct ip *);  
950         ip->ip_len = htons((u_short) ip->ip_len);  
951         ip->ip_off = htons((u_short) ip->ip_off);  
952         ip->ip_sum = 0;  
953         ip->ip_sum = in_cksum(copym, ip->ip_hl << 2);  
954         (void) looutput(ifp, copym, (struct sockaddr *) dst, NULL);  
955     }  
956 }
```

ip\_output.c

## Duplicate and queue packet

929-956

Copying the packet isn't enough; the packet must look as though it was received on the output interface, so ip\_mloopback converts ip\_len and ip\_off to network byte order and computes the checksum for the packet. looutput takes care of putting the packet on the IP input queue.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.16 Performance Considerations

The multicast implementation in Net/3 has several potential performance bottlenecks. Since many Ethernet cards do not support perfect filtering of multicast addresses, the operating system must be prepared to discard multicast packets that pass through the hardware filter. In the worst case, an Ethernet card may fall back to receiving all multicast packets, most of which must be discarded by ipintr when they are found not to contain a valid IP multicast group address.

IP uses a simple linear list and linear search to filter incoming IP datagrams. If

the list grows to any appreciable length, a caching mechanism such as moving the most recently received address to the front of the list would help performance.

---

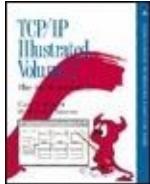
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 12. IP Multicasting

### 12.17 Summary

In this chapter we described how a single host processes IP multicast datagrams. We looked at the format of an IP class D address and an Ethernet multicast address and the mapping between the two.

We discussed the `in_multi` and `ether_multi` structures, and we saw that each IP multicast interface maintains its own group membership list and that each Ethernet interface maintains a list of Ethernet multicast addresses.

During input processing, IP multicasts are accepted only if they arrive on an interface that is a member of their destination

group, although they may be forwarded to other interfaces if the system is configured as a multicast router.

Systems configured as multicast routers must accept all multicast packets on every interface. This can be done quickly by issuing the SIOCADDMULTI command for the INADDR\_ANY address.

The ip\_moptions structure is the cornerstone of multicast output processing. It controls the selection of an output interface, the TTL field of the multicast datagram, and the loopback policy. It also holds references to the in\_multi structures, which determine when an interface joins or leaves an IP multicast group.

We also discussed the two concepts implemented by the multicast TTL value: packet lifetime and packet scope.

## Exercises

What is the difference between

sending an IP broadcast packet to  
**12.1** 255.255.255.255 and sending an IP multicast to the all-hosts group 224.0.0.1?

Why are interfaces identified by their IP unicast addresses in the multicasting code? What must be  
**12.2** changed so that an interface could send and receive multicast datagrams but not have a unicast IP address?

In [Section 12.3](#) we said that 32 IP groups are mapped to a single Ethernet address. Since 9 bits of a  
**12.3** 32-bit address are not included in the mapping, why didn't we say that  $512 (2^9)$  IP groups mapped to a single Ethernet address?

Why do you think  
IP\_MAX\_MEMBERSHIPS is set to 20? Could it be set to a larger  
**12.4** value? Hint: Consider the size of

the ip\_moptions structure (Figure 12.15).

What happens when a multicast datagram is looped back by IP and 12.5 is also received by the hardware interface on which it is transmitted (i.e., a nonsimplex interface)?

Draw a picture of a network with a multihomed host so that a multicast packet sent on one interface may 12.6 be received on the other interface even if the host is not acting as a multicast router.

Trace the membership add request 12.7 through the SLIP and loopback interfaces instead of the Ethernet interface.

How could a process request that 12.8 the kernel join more than IP\_MAX\_MEMBERSHIPS?

Computing the checksum on a looped back packet is superfluous.

**12.9** Design a method to avoid the checksum computation for loopback packets.

**12.10** How many IP multicast groups could an interface join without reusing an Ethernet multicast address?

**12.11** The careful reader might have noticed that `in_delmulti` assumes that the interface has defined an `ioctl` function when it issues the `SIOCDELMULTI` request. Why is this OK?

**12.12** What happens to the mbuf allocated in `ip_getmoptions` if an unrecognized option is requested?

Why is the group membership mechanism separate from the

## **12.13** binding mechanism used to receive unicast and broadcast datagrams?

---

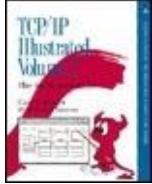
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 13. IGMP: Internet Group Management Protocol

[Section 13.1. Introduction](#)

[Section 13.2. Code Introduction](#)

[Section 13.3. igmp Structure](#)

[Section 13.4. IGMP protosw Structure](#)

[Section 13.5. Joining a Group:  
igmp\\_joingroup Function](#)

[Section 13.6. igmp\\_fasttim0 Function](#)

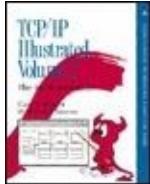
[Section 13.7. Input Processing:  
igmp\\_input Function](#)

[Section 13.8. Leaving a Group:  
igmp\\_leavegroup Function](#)

[Section 13.9. Summary](#)



Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.1 Introduction

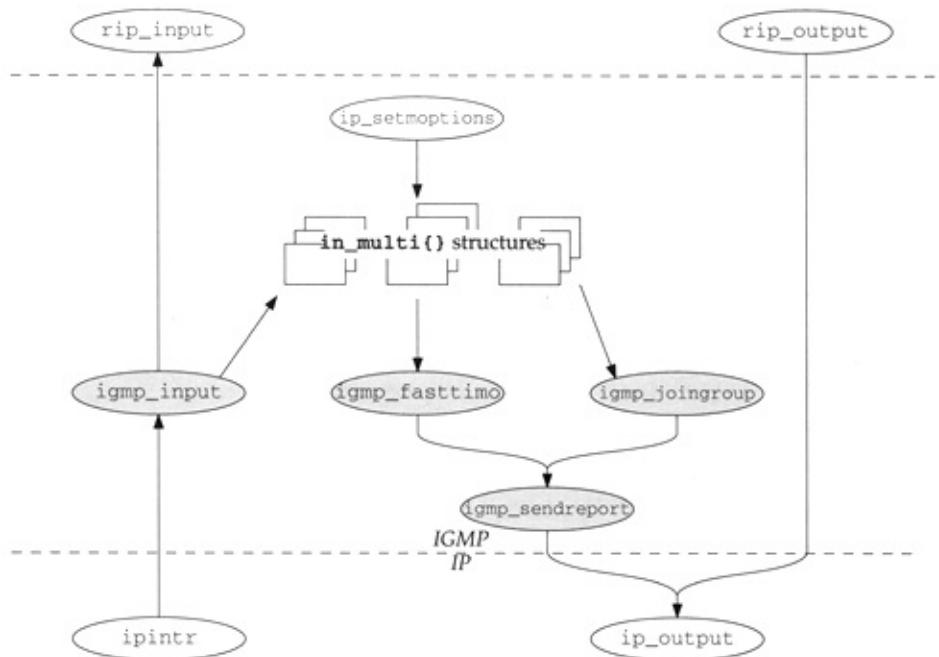
IGMP conveys group membership information between hosts and routers on a local network. Routers periodically multicast IGMP queries to the all-hosts group. Hosts respond to the queries by multicasting IGMP report messages. The IGMP specification appears in RFC 1112. Chapter 13 of Volume 1 describes the specification of IGMP and provides some examples.

From an architecture perspective, IGMP is a transport protocol above IP. It has a protocol number (2) and its messages are carried in IP datagrams (as with ICMP).

IGMP usually isn't accessed directly by a process but, as with ICMP, a process can send and receive IGMP messages through an IGMP socket. This feature enables multicast routing daemons to be implemented as user-level processes.

Figure 13.1 shows the overall organization of the IGMP protocol in Net/3.

## Figure 13.1. Summary of IGMP processing.



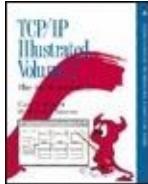
The key to IGMP processing is the collection of `in_multi` structures shown in

the center of [Figure 13.1](#). An incoming IGMP query causes `igmp_input` to initialize a countdown timer for each `in_multi` structure. The timers are updated by `igmp_fasttim`, which calls `igmp_sendreport` as each timer expires.

We saw in [Chapter 12](#) that `ip_setmoptions` calls `igmp_joingroup` when a new `in_multi` structure is created. `igmp_joingroup` calls `igmp_sendreport` to announce the new group and enables the group's timer to schedule a second announcement a short time later. `igmp_sendreport` takes care of formatting an IGMP message and passing it to `ip_output`.

On the left and right of [Figure 13.1](#) we see that a raw socket can send and receive IGMP messages directly.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.2 Code Introduction

The IGMP protocol is implemented in four files listed in [Figure 13.2](#).

**Figure 13.2. Files discussed in this chapter.**

File	Description
netinet/igmp.h	IGMP protocol definitions
netinet/igmp_var.h	IGMP implementation definitions
netinet/in_var.h	IP multicast data structures
netinet/igmp.c	IGMP protocol implementation

### Global Variables

Three new global variables, shown in

Figure 13.3, are introduced in this chapter.

## Figure 13.3. Global variables introduced in this chapter.

Variable	Datatype	Description
igmp_all_hosts_group	u_long	all-hosts group address in network byte order
igmp_timers_are_running	int	true if any IGMP timer is active, false otherwise
igmpstat	struct igmpstat	IGMP statistics (Figure 13.4).

## Statistics

IGMP statistics are maintained in the igmpstat variables shown in Figure 13.4.

## Figure 13.4. IGMP statistics.

igmpstat member	Description
igps_rcv_badqueries	#messages received as invalid queries
igps_rcv_badreports	#messages received as invalid reports
igps_rcv_badsum	#messages received with bad checksum
igps_rcv_ourreports	#messages received as reports for local groups
igps_rcv_queries	#messages received as membership queries
igps_rcv_reports	#messages received as membership reports
igps_rcv_tooshort	#messages received with too few bytes
igps_rcv_total	total #IGMP messages received
igps_snd_reports	#messages sent as membership reports

Figure 13.5 shows some sample output of these statistics, from the netstat -p igmp

command on vangogh.cs.berkeley.edu.

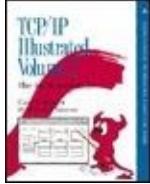
### Figure 13.5. Sample IGMP statistics.

netstat -p igmp output	igmpstat member
18774 messages received	igps_rcv_total
0 messages received with too few bytes	igps_rcv_tooshort
0 messages received with bad checksum	igps_rcv_badsum
18774 membership queries received	igps_rcv_queries
0 membership queries received with invalid field(s)	igps_rcv_badqueries
0 membership reports received	igps_rcv_reports
0 membership reports received with invalid field(s)	igps_rcv_badreports
0 membership reports received for groups to which we belong	igps_rcv_ourreports
0 membership reports sent	igps_snd_reports

From Figure 13.5 we can tell that vangogh is attached to a network where IGMP is being used, but that vangogh is not joining any multicast groups, since igps\_snd\_reports is 0.

## SNMP Variables

There is no standard SNMP MIB for IGMP, but [McCloghrie and Farinacci 1994a] describes an experimental MIB for IGMP.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.3 igmp Structure

An IGMP message is only 8 bytes long.  
[Figure 13.6](#) shows the igmp structure used by Net/3.

**Figure 13.6. igmp structure.**

---

```
43 struct igmp {  
44     u_char    igmp_type;          /* version & type of IGMP message */  
45     u_char    igmp_code;         /* unused, should be zero */  
46     u_short   igmp_cksum;        /* IP-style checksum */  
47     struct in_addr igmp_group;  /* group address being reported */  
48 };
```

*- igmp.h*

43-44

A 4-bit version code and a 4-bit type code are contained within igmp\_type. [Figure](#)

[13.7](#) shows the standard values.

## Figure 13.7. IGMP message types.

Version	Type	igmp_type	Description
1	1	0x11 (IGMP_HOST_MEMBERSHIP_QUERY)	membership query
1	2	0x12 (IGMP_HOST_MEMBERSHIP_REPORT)	membership report
1	3	0x13	DVMRP message (Chapter 14)

Only version 1 messages are used by Net/3. Multicast routers send type 1 (IGMP\_HOST\_MEMBERSHIP\_QUERY) messages to solicit membership reports from hosts on the local network. The response to a type 1 IGMP message is a type 2 (IGMP\_HOST\_MEMBERSHIP\_REPORT) message from the hosts reporting their multicast membership information. Type 3 messages transport multicast routing information between routers ([Chapter 14](#)). A host never processes type 3 messages. The remainder of this chapter discusses only type 1 and 2 messages.

45-46

igmp\_code is unused in IGMP version 1, and igmp\_cksum is the familiar IP

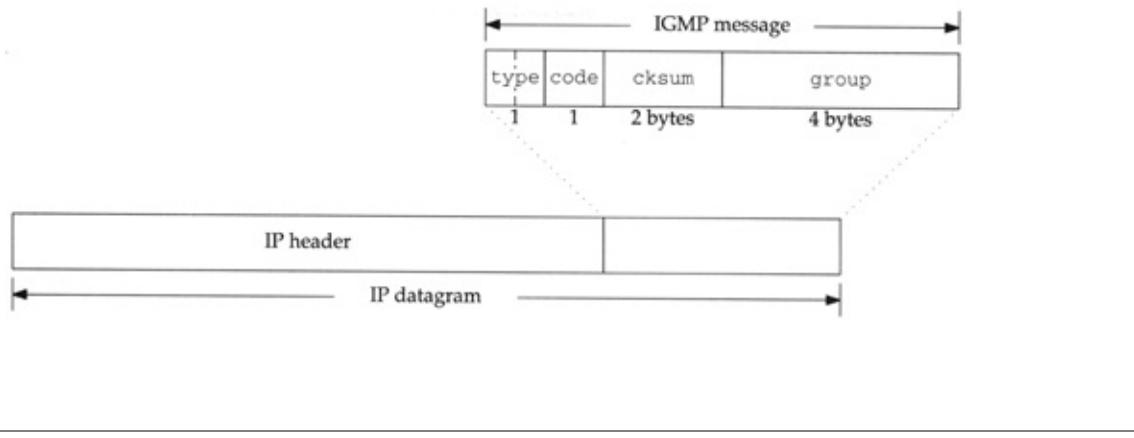
checksum computed over all 8 bytes of the IGMP message.

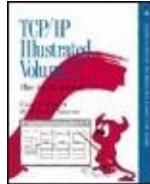
47-48

igmp\_group is 0 for queries. For replies, it contains the multicast group being reported.

Figure 13.8 shows the structure of an IGMP message relative to an IP datagram.

### Figure 13.8. An IGMP message (igmp\_omitted).





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.4 IGMP protosw Structure

Figure 13.9 describes the protosw structure for IGMP.

#### Figure 13.9. The IGMP protosw structure.

Member	inetsw[5]	Description
pr_type	<code>SOCK_RAW</code>	IGMP provides raw packet services
pr_domain	<code>&amp;inetdomain</code>	IGMP is part of the Internet domain
pr_protocol	<code>IPPROTO_IGMP (2)</code>	appears in the <code>ip_p</code> field of the IP header
pr_flags	<code>PR_ATOMIC PR_ADDR</code>	socket layer flags, not used by protocol processing
pr_input	<code>igmp_input</code>	receives messages from IP layer
pr_output	<code>rip_output</code>	sends IGMP message to IP layer
pr_ctlinput	<code>0</code>	not used by IGMP
pr_ctloutput	<code>rip_ctloutput</code>	respond to administrative requests from a process
pr_usrreq	<code>rip_usrreq</code>	respond to communication requests from a process
pr_init	<code>igmp_init</code>	initialization for IGMP
pr_fasttimo	<code>igmp_fasttimo</code>	process pending membership reports
pr_slowtimo	<code>0</code>	not used by IGMP
pr_drain	<code>0</code>	not used by IGMP
pr_sysctl	<code>0</code>	not used by IGMP

Although it is possible for a process to

send raw IP packets through the IGMP protosw entry, in this chapter we are concerned only with how the kernel processes IGMP messages. [Chapter 32](#) discusses how a process can access IGMP using a raw socket.

There are three events that trigger IGMP processing:

- a local interface has joined a new multicast group ([Section 13.5](#)),
- an IGMP timer has expired ([Section 13.6](#)), and
- an IGMP query is received ([Section 13.7](#)).

There are also two events that trigger local IGMP processing but do not result in any messages being sent:

- an IGMP report is received ([Section 13.7](#)), and
- a local interface leaves a multicast group ([Section 13.8](#)).

These five events are discussed in the following sections.

---

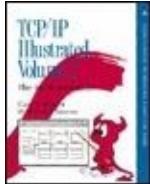
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.5 Joining a Group: `igmp_joingroup` Function

We saw in [Chapter 12](#) that `igmp_joingroup` is called by `in_addmulti` when a new `in_multi` structure is created. Subsequent requests to join the same group only increase the reference count in the `in_multi` structure; `igmp_joingroup` is not called. `igmp_joingroup` is shown in [Figure 13.10](#)

**Figure 13.10. `igmp_joingroup` function.**

```
164 void
165 igmp_joingroup(inm)
166 struct in_multi *inm;
167 {
168     int     s = splnet();
169     if (inm->inm_addr.s_addr == igmp_all_hosts_group ||
170         inm->inm_ifp == &loif)
171         inm->inm_timer = 0;
172     else {
173         igmp_sendreport(inm);
174         inm->inm_timer = IGMP_RANDOM_DELAY(inm->inm_addr);
175         igmp_timers_are_running = 1;
176     }
177     splx(s);
178 }
```

-igmp.c

## 164-178

inm points to the new in\_multi structure for the group. If the new group is the all-hosts group, or the membership request is for the loopback interface, inm\_timer is disabled and igmp\_joingroup returns. Membership in the all-hosts group is never reported, since every multicast host is assumed to be a member of the group. Sending a membership report to the loopback interface is unnecessary, since the local host is the only system on the loopback network and it already knows its membership status.

In the remaining cases, a report is sent immediately for the new group, and the group timer is set to a random value based on the group. The global flag

`igmp_timers_are_running` is set to indicate that at least one timer is enabled. `igmp_fasttim` (Section 13.6) examines this variable to avoid unnecessary processing.

When the timer for the new group expires, a second membership report is issued. The duplicate report is harmless, but it provides insurance in case the first report is lost or damaged. The report delay is computed by `IGMP_RANDOM_DELAY` (Figure 13.11).

### Figure 13.11. `IGMP_RANDOM_DELAY` function.

```
59 /* igmp_var.h
60 * Macro to compute a random timer value between 1 and (IGMP_MAX_REPORTING_
61 * DELAY * countdown frequency). We generate a "random" number by adding
62 * the total number of IP packets received, our primary IP address, and the
63 * multicast address being timed-out. The 4.3 random() routine really
64 * ought to be available in the kernel!
65 */
66 #define IGMP_RANDOM_DELAY(multiaddr) \
67     /* struct in_addr multiaddr; */ \
68     (ipstat.ipi_total + \
69      ntohs(IA_SIN(in_ifaddr)->sin_addr.s_addr) + \
70      ntohs((multiaddr).s_addr) \
71      ) \
72      % (IGMP_MAX_HOST_REPORT_DELAY * PR_FASTHZ) + 1 \
73 }
```

According to RFC 1122, report timers should be set to a random time between 0 and 10 (IGMP\_MAX\_HOST\_REPORT\_DELAY) seconds. Since IGMP timers are decremented five (PR\_FASTHZ) times per second, IGMP\_RANDOM\_DELAY must pick a random value between 1 and 50. If  $r$  is the random number computed by adding the total number of IP packets received, the host's primary IP address, and the multicast group, then

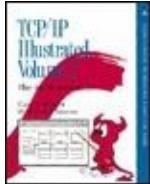
$$0 \leq (r \bmod 50) \leq 49$$

and

$$1 \leq (r \bmod 50) + 1 \leq 50$$

Zero is avoided because it would disable the timer and no report would be sent.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.6 igmp\_fasttimo Function

Before looking at igmp\_fasttimo, we need to describe the mechanism used to traverse the in\_multi structures.

To locate each in\_multi structure, Net/3 must traverse the in\_multi list for each interface. During a traversal, an in\_multistep structure (shown in [Figure 13.12](#)) records the position.

#### Figure 13.12. in\_multistep function.

```
123 struct in_multistep {  
124     struct in_ifaddr *i_ia;  
125     struct in_multi *i_inm;  
126 };
```

## 123-126

i\_ia points to the *next* in\_ifaddr interface structure and i\_inm points to the *next* in\_multi structure for the *current* interface.

The IN\_FIRST\_MULTI and IN\_NEXT\_MULTI macros (shown in [Figure 13.13](#)) traverse the lists.

**Figure 13.13. IN\_FIRST\_MULTI and IN\_NEXT\_MULTI structures.**

```
-----in_var.h
147 /*
148  * Macro to step through all of the in_multi records, one at a time.
149  * The current position is remembered in "step", which the caller must
150  * provide. IN_FIRST_MULTI(), below, must be called to initialize "step"
151  * and get the first record. Both macros return a NULL "inm" when there
152  * are no remaining records.
153 */
154 #define IN_NEXT_MULTI(step, inm) \
155     /* struct in_multistep step; */ \
156     /* struct in_multi *inm; */ \
157 { \
158     if (((inm) = (step).i_inm) != NULL) \
159         (step).i_inm = (inm)->inm_next; \
160     else \
161         while ((step).i_ia != NULL) { \
162             (inm) = (step).i_ia->ia_multiaddrs; \
163             (step).i_ia = (step).i_ia->ia_next; \
164             if ((inm) != NULL) { \
165                 (step).i_inm = (inm)->inm_next; \
166                 break; \
167             } \
168         } \
169 }
170 #define IN_FIRST_MULTI(step, inm) \
171     /* struct in_multistep step; */ \
172     /* struct in_multi *inm; */ \
173 { \
174     (step).i_ia = in_ifaddr; \
175     (step).i_inm = NULL; \
176     IN_NEXT_MULTI((step), (inm)); \
177 }
```

-----in\_var.h

154-169

If the in\_multi list has more entries, i\_inm is advanced to the next entry. When IN\_NEXT\_MULTI reaches the end of a multicast list, i\_ia is advanced to the next interface and i\_inm to the first in\_multi structure associated with the interface. If the interface has no multicast structures, the while loop continues to advance through the interface list until all interfaces have been searched.

170-177

The in\_multistep array is initialized to point to the first in\_ifaddr structure in the in\_ifaddr list and i\_inm is set to null. IN\_NEXT\_MULTI finds the first in\_multi structure.

We know from [Figure 13.9](#) that igmp\_fasttimmo is the fast timeout function for IGMP and is called five times per second. igmp\_fasttimmo (shown in [Figure 13.14](#)) decrements multicast report timers and sends a report when the timer expires.

## Figure 13.14. igmp\_fasttimmo function.

```
187 void - igmp.c
188 igmp_fasttimmo()
189 {
190     struct in_multi *inm;
191     int     s;
192     struct in_multistep step;
193     /*
194      * Quick check to see if any work needs to be done, in order
195      * to minimize the overhead of fasttimmo processing.
196      */
197     if (!igmp_timers_are_running)
198         return;
199     s = splnet();
200     igmp_timers_are_running = 0;
201     IN_FIRST_MULTI(step, inm);
202     while (inm != NULL) {
203         if (inm->inm_timer == 0) {
204             /* do nothing */
205         } else if (--inm->inm_timer == 0) {
206             igmp_sendreport(inm);
207         } else {
208             igmp_timers_are_running = 1;
209         }
210         IN_NEXT_MULTI(step, inm);
211     }
212     splx(s);
213 }
```

187-198

If igmp\_timers\_are\_running is false, igmp\_fasttimmo returns immediately instead of wasting time examining each timer.

199-213

igmp\_fasttimmo resets the running flag and then initializes step and inm with IN\_FIRST\_MULTI. The igmp\_fasttimmo function locates each in\_multi structure with the while loop and the

IN\_NEXT\_MULTI macro. For each structure:

- If the timer is 0, there is nothing to be done.
- If the timer is nonzero, it is decremented. If it reaches 0, an IGMP membership report is sent for the group.
- If the timer is still nonzero, then at least one timer is still running, so igmp\_timers\_are\_running is set to 1.

## igmp\_sendreport Function

The igmp\_sendreport function (shown in [Figure 13.15](#)) constructs and sends an IGMP report message for a single multicast group.

**Figure 13.15. igmp\_sendreport function.**

---

```

214 static void
215 igmp_sendreport(inm)
216 struct in_multi *inm;
217 {
218     struct mbuf *m;
219     struct igmp *igmp;
220     struct ip *ip;
221     struct ip_moptions *imo;
222     struct ip_moptions simo;
223
224     MGETHDR(m, M_DONTWAIT, MT_HEADER);
225     if (m == NULL)
226         return;
227     /*
228      * Assume max_linkhdr + sizeof(struct ip) + IGMP_MINLEN
229      * is smaller than mbuf size returned by MGETHDR.
230      */
231     m->m_data += max_linkhdr;
232     m->m_len = sizeof(struct ip) + IGMP_MINLEN;
233     m->m_pkthdr.len = sizeof(struct ip) + IGMP_MINLEN;
234
235     ip = mtod(m, struct ip *);
236     ip->ip_tos = 0;
237     ip->ip_len = sizeof(struct ip) + IGMP_MINLEN;
238     ip->ip_off = 0;
239     ip->ip_p = IPPROTO_IGMP;
240     ip->ip_src.s_addr = INADDR_ANY;
241     ip->ip_dst = inm->inm_addr;
242
243     igmp = (struct igmp *) (ip + 1);
244     igmp->igmp_type = IGMP_HOST_MEMBERSHIP_REPORT;
245     igmp->igmp_code = 0;
246     igmp->igmp_group = inm->inm_addr;
247     igmp->igmp_cksum = 0;
248     igmp->igmp_cksum = in_cksum(m, IGMP_MINLEN);
249
250     imo = &simo;
251     bzero((caddr_t) imo, sizeof(*imo));
252     imo->imo_multicast_ifp = inm->inm_ifp;
253     imo->imo_multicast_ttl = 1;
254
255     /*
256      * Request loopback of the report if we are acting as a multicast
257      * router, so that the process-level routing demon can hear it.
258      */
259     extern struct socket *ip_mrouter;
260     imo->imo_multicast_loop = (ip_mrouter != NULL);
261
262     ip_output(m, NULL, NULL, 0, imo);
263
264     ++igmpstat.igps_snd_reports;
265 }

```

---

igmp.c

## 214-232

The single argument inm points to the in\_multi structure for the group being reported. igmp\_sendreport allocates a new mbuf and prepares it for an IGMP message. igmp\_sendreport leaves room

for a link-layer header and sets the length of the mbuf and packet to the length of an IGMP message.

233-245

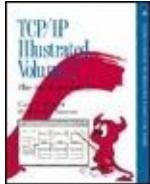
The IP header and IGMP message is constructed one field at a time. The source address for the datagram is set to INADDR\_ANY, and the destination address is the multicast group being reported. ip\_output replaces INADDR\_ANY with the unicast address of the outgoing interface. Every member of the group receives the report as does every multicast router (since multicast routers receive *all* IP multicasts).

246-260

Finally, igmp\_sendreport constructs an ip\_moptions structure to go along with the message sent to ip\_output. The interface associated with the in\_multi structure is selected as the outgoing interface; the TTL is set to 1 to keep the report on the local network; and, if the local system is configured as a router, multicast loopback

is enabled for this request.

The process-level multicast router must hear the membership reports. In [Section 12.14](#) we saw that IGMP datagrams are always accepted when the system is configured as a multicast router. Through the normal transport demultiplexing code, the messages are passed to `igmp_input`, the `pr_input` function for IGMP ([Figure 13.9](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.7 Input Processing: `igmp_input` Function

In [Section 12.14](#) we described the multicast processing portion of `ipintr`. We saw that a multicast router accepts *any* IGMP message, but a multicast host accepts only IGMP messages that arrive on an interface that is a member of the destination multicast group (i.e., queries and membership reports for which the receiving interface is a member).

The accepted messages are passed to `igmp_input` by the standard protocol demultiplexing mechanism. The beginning and end of `igmp_input` are shown in [Figure](#)

[13.16.](#) The code for each IGMP message type is described in following sections.

**Figure 13.16. igmp\_input function.**

```

52 void
53 igmp_input(m, iphlen)
54 struct mbuf *m;
55 int     iphlen;
56 {
57     struct igmp *igmp;
58     struct ip *ip;
59     int      igmplen;
60     struct ifnet *ifp = m->m_pkthdr.rcvif;
61     int      minlen;
62     struct in_multi *inm;
63     struct in_ifaddr *ia;
64     struct in_multistep step;
65     ++igmpstat.igps_rcv_total;
66     ip = mtod(m, struct ip *);
67     igmplen = ip->ip_len;
68     /*
69      * Validate lengths
70      */
71     if (igmplen < IGMP_MINLEN) {
72         ++igmpstat.igps_rcv_tooshort;
73         m_freem(m);
74         return;
75     }
76     minlen = iphlen + IGMP_MINLEN;
77     if ((m->m_flags & M_EXT || m->m_len < minlen) &&
78         (m = m_pullup(m, minlen)) == 0) {
79         ++igmpstat.igps_rcv_tooshort;
80         return;
81     }
82     /*
83      * Validate checksum
84      */
85     m->m_data += iphlen;
86     m->m_len -= iphlen;
87     igmp = mtod(m, struct igmp *);
88     if (in_cksum(m, igmplen)) {
89         ++igmpstat.igps_rcv_badsum;
90         m_freem(m);
91         return;
92     }
93     m->m_data -= iphlen;
94     m->m_len += iphlen;
95     ip = mtod(m, struct ip *);
96     switch (igmp->igmp_type) {
97
98         /* switch cases */
99
100    }
101
102    /*
103     * Pass all valid IGMP packets up to any process(es) listening
104     * on a raw IGMP socket.
105     */
106    rip_input(m);
107 }

```

## Validate IGMP message

## 52-96

The function ipintr passes m, a pointer to the received packet (stored in an mbuf), and iphlen, the size of the IP header in the datagram.

The datagram must be large enough to contain an IGMP message (IGMP\_MINLEN), must be contained within a standard mbuf header (m\_pullup), and must have a correct IGMP checksum. If any errors are found, they are counted, the datagram is silently discarded, and igmp\_input returns.

The body of igmp\_input processes the validated messages based on the code in igmp\_type. Remember from [Figure 13.6](#) that igmp\_type includes a version code and a type code. The switch statement is based on the combined value stored in igmp\_type ([Figure 13.7](#)). Each case is described separately in the following sections.

## Pass IGMP messages to raw IP

157-163

There is no default case for the switch statement. Any valid message (i.e., one that is properly formed) is passed to rip\_input where it is delivered to any process listening for IGMP messages. IGMP messages with versions or types that are unrecognized by the kernel can be processed or discarded by the listening processes.

The mrouted program depends on this call to rip\_input so that it receives membership queries and reports.

## **Membership Query: IGMP\_HOST\_MEMBERSHIP\_QUERY**

RFC 1075 recommends that multicast routers issue an IGMP membership query at least once every 120 seconds. The query is sent to group 224.0.0.1 (the all-hosts group). [Figure 13.17](#) shows how the message is processed by a host.

## Figure 13.17. Input processing of the IGMP query message.

```
97     case IGMP_HOST_MEMBERSHIP_QUERY:
98         ++igmpstat.igps_rcv_queries;
99
100        if (ifp == &loif)
101            break;
102
103        if (ip->ip_dst.s_addr != igmp_all_hosts_group) {
104            ++igmpstat.igps_rcv_badqueries;
105            m_freem(m);
106            return;
107        }
108        /*
109         * Start the timers in all of our membership records for
110         * the interface on which the query arrived, except those
111         * that are already running and those that belong to the
112         * "all-hosts" group.
113         */
114        IN_FIRST_MULTI(step, inm);
115        while (inm != NULL) {
116            if (inm->inm_ifp == ifp && inm->inm_timer == 0 &&
117                inm->inm_addr.s_addr != igmp_all_hosts_group) {
118                inm->inm_timer =
119                    IGMP_RANDOM_DELAY(inm->inm_addr);
120                igmp_timers_are_running = 1;
121            }
122            IN_NEXT_MULTI(step, inm);
123        }
124    break;
```

igmp.c

97-122

Queries that arrive on the loopback interface are silently discarded ([Exercise 13.1](#)). Queries by definition are sent to the all-hosts group. If a query arrives addressed to a different address, it is counted in igps\_rcv\_badqueries and discarded.

The receipt of a query message does not trigger an immediate flurry of IGMP

membership reports. Instead, `igmp_input` resets the membership timers for each group associated with the interface on which the query was received to a random value with `IGMP_RANDOM_DELAY`. When the timer for a group expires, `igmp_fasttimo` sends a membership report. Meanwhile, the same activity is occurring on all the other hosts that received the IGMP query. As soon as the random timer for a particular group expires on one host, it is multicast to that group. This report cancels the timers on the other hosts so that only one report is multicast to the network. The routers, as well as any other members of the group, receive the report.

The one exception to this scenario is the all-hosts group. A timer is never set for this group and a report is never sent.

## **Membership Report: IGMP\_HOST\_MEMBERSHIP\_REPORT**

The receipt of an IGMP membership report is one of the two events we mentioned in [Section 13.1](#) that does not result in an

IGMP message. The effect of the message is local to the interface on which it was received. [Figure 13.18](#) shows the message processing.

## Figure 13.18. Input processing of the IGMP report message.

```
123     case IGMP_HOST_MEMBERSHIP_REPORT:                                igmp.c
124         ++igmpstat.igps_rcv_reports;
125
126         if (ifp == &loif)
127             break;
128
129         if (!IN_MULTICAST(ntohl(igmp->igmp_group.s_addr)) ||
130             igmp->igmp_group.s_addr != ip->ip_dst.s_addr) {
131             ++igmpstat.igps_rcv_badreports;
132             m_freem(m);
133             return;
134         }
135         /*
136          * KLUDGE: if the IP source address of the report has an
137          * unspecified (i.e., zero) subnet number, as is allowed for
138          * a booting host, replace it with the correct subnet number
139          * so that a process-level multicast routing demon can
140          * determine which subnet it arrived from. This is necessary
141          * to compensate for the lack of any way for a process to
142          * determine the arrival interface of an incoming packet.
143
144         if ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) == 0) {
145             IIP_TO_IA(ifp, ia);
146             if (ia)
147                 ip->ip_src.s_addr = htonl(ia->ia_subnet);
148
149             /*
150              * If we belong to the group being reported, stop
151              * our timer for that group.
152
153             IN_LOOKUP_MULTI(igmp->igmp_group, ifp, inm);
154             if (inm != NULL) {
155                 inm->inm_timer = 0;
156                 ++igmpstat.igps_rcv_ourreports;
157             }
158         }
159     break;
```

123-156

Reports sent to the loopback interface are

discarded, as are membership reports sent to the incorrect multicast group. That is, the message must be addressed to the group identified within the message.

The source address of an incompletely initialized host might not include a network or host number (or both). `igmp_report` looks at the class A network portion of the address, which can only be 0 when the network and subnet portions of the address are 0. If this is the case, the source address is set to the subnet address, which includes the network ID and subnet ID, of the receiving interface. The only reason for doing this is to inform a process-level daemon of the receiving interface, which is identified by the subnet number.

If the receiving interface belongs to the group being reported, the associated report timer is reset to 0. In this way the first report sent to the group stops any other hosts from issuing a report. It is only necessary for the router to know that at least one interface on the network is a member of the group. The router does not

need to maintain an explicit membership list or even a counter.

---

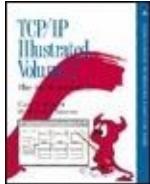
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.8 Leaving a Group: `igmp_leavegroup` Function

We saw in [Chapter 12](#) that `in_delmulti` calls `igmp_leavegroup` when the last reference count in the associated `in_multi` structure drops to 0.

#### Figure 13.19. `igmp_leavegroup` function.

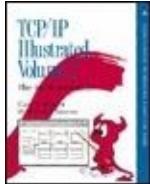
```
179 void igmp_leavegroup(inm) igmp.c
180 struct in_multi *inm;
181 {
182     /*
183      * No action required on leaving a group.
184      */
185 }
186 }
```

179-186

As we can see, IGMP takes no action when an interface leaves a group. No explicit notification is sent the next time a multicast router issues an IGMP query, the interface does not generate an IGMP report for this group. If no report is generated for a group, the multicast router assumes that all the interfaces have left the group and stops forwarding multicast packets for the group to the network.

If the interface leaves the group while a report is pending (i.e., the group's report timer is running), the report is never sent, since the timer is discarded by `in_delmulti` ([Figure 12.36](#)) along with the `in_multi` structure for the group when `icmp_leavegroup` returns.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 13. IGMP: Internet Group Management Protocol

### 13.9 Summary

In this chapter we described IGMP, which communicates IP multicast membership information between hosts and routers on a single network. IGMP membership reports are generated when an interface joins a group, and on demand when multicast routers issue an IGMP report query message.

The design of IGMP minimizes the number of messages required to communicate membership information:

- Hosts announce their membership when they join a group.

- Response to membership queries are delayed for a random interval, and the first response suppresses any others.
- Hosts are silent when they leave a group.
- Membership queries are sent no more than once per minute.

Multicast routers share the IGMP information they collect with each other ([Chapter 14](#)) to route multicast datagrams toward remote members of the multicast destination group.

## Exercises

Why isn't it necessary to respond to **13.1** an IGMP query on the loopback interface?

**13.2** Verify the assumption stated on lines 226 to 229 in [Figure 13.15](#).

Is it necessary to set random delays  
**13.3** for membership queries that arrive  
on a point-to-point network  
interface?

---

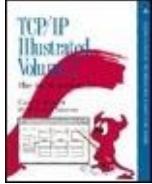
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 14. IP Multicast Routing

[Section 14.1. Introduction](#)

[Section 14.2. Code Introduction](#)

[Section 14.3. Multicast Output Processing Revisited](#)

[Section 14.4. mrouted Daemon](#)

[Section 14.5. Virtual Interfaces](#)

[Section 14.6. IGMP Revisited](#)

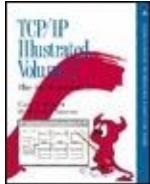
[Section 14.7. Multicast Routing](#)

[Section 14.8. Multicast Forwarding: ip\\_mforward Function](#)

[Section 14.9. Cleanup: ip\\_mrouter\\_done Function](#)

[Section 14.10. Summary](#)

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.1 Introduction

The previous two chapters discussed multicasting on a single network. In this chapter we look at multicasting across an entire internet. We describe the operation of the mrouted program, which computes the multicast routing tables, and the kernel functions that forward multicast datagrams between networks.

Technically, multicast *packets* are forwarded. In this chapter we assume that every multicast packet contains an entire datagram (i.e., there are no fragments), so we use the term *datagram* exclusively. Net/3 forwards IP fragments as well as IP datagrams.

**Figure 14.1** shows several versions of mrouted and how they correspond to the BSD releases. The mrouted releases include both the user-level daemons and the kernel-level multicast code.

## **Figure 14.1. mrouted and IP multicasting releases.**

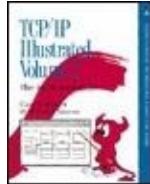
mrouted version	Description
1.2	modifies the 4.3BSD Tahoe release included with 4.4BSD and Net/3
2.0	
3.3	modifies SunOS 4.1.3

IP multicast technology is an active area of research and development. This chapter discusses version 2.0 of the multicast software, which is included in Net/3 but is considered an obsolete implementation. Version 3.3 was released too late to be discussed fully in this text, but we will point out various 3.3 features along the way.

Because commercial multicast routers are not widely deployed, multicast networks are often constructed using multicast

*tunnels*, which connect two multicast routers over a standard IP unicast internet. Multicast tunnels are supported by Net/3 and are constructed with the Loose Source Record Route (LSRR) option ([Section 9.6](#)). An improved tunneling technique encapsulates the IP multicast datagram within an IP unicast datagram and is supported by version 3.3 of the multicast code but is not supported by Net/3.

As in [Chapter 12](#), we use the generic term *transport protocols* to refer to the protocols that send and receive multicast datagrams, but UDP is the only Internet protocol that supports multicasting.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.2 Code Introduction

The three files listed in [Figure 14.2](#) are discussed in this chapter.

**Figure 14.2. Files discussed in this chapter.**

File	Description
netinet/ip_mroute.h	multicast structure definitions
netinet/ip_mroute.c	multicast routing functions
netinet/raw_ip.c	multicast routing options

### Global Variables

The global variables used by the multicast routing code are shown in [Figure 14.3](#).

## Figure 14.3. Global variables introduced in this chapter.

Variable	Datatype	Description
cached_mrt	struct mrt	one-behind cache for multicast routing
cached_origin	u_long	multicast group for one-behind cache
cached_originmask	u_long	mask for multicast group for one-behind cache
mrtstat	struct mrtstat	multicast routing statistics
mrttable	struct mrt *[]	hash table of pointers to multicast routes
numvifs	vifi_t	number of enabled multicast interfaces
viftable	struct vif[]	array of virtual multicast interfaces

## Statistics

All the statistics collected by the multicast routing code are found in the mrtstat structure described by [Figure 14.4](#). [Figure 14.5](#) shows some sample output of these statistics, from the netstat -gs command.

## Figure 14.4. Statistics collected in this chapter.

mrtstat member	Description	Used by SNMP
mrts_mrt_lookups	#multicast route lookups	
mrts_mrt_misses	#multicast route cache misses	
mrts_grp_lookups	#group address lookups	
mrts_grp_misses	#group address cache misses	
mrts_no_route	#multicast route lookup failures	
mrts_bad_tunnel	#packets with malformed tunnel options	
mrts_cant_tunnel	#packets with no room for tunnel options	

## Figure 14.5. Sample IP multicast routing statistics.

netstat -gs output	mrtstat members
multicast routing:	
329569328 multicast route lookups	mrts_mrt_lookups
9377023 multicast route cache misses	mrts_mrt_misses
242754062 group address lookups	mrts_grp_lookups
159317788 group address cache misses	mrts_grp_misses
65648 datagrams with no route for origin	mrts_no_route
0 datagrams with malformed tunnel options	mrts_bad_tunnel
0 datagrams with no room for tunnel options	mrts_cant_tunnel

These statistics are from a system with two physical interfaces and one tunnel interface. These statistics show that the multicast route is found in the cache 98% of the time. The group address cache is less effective with only a 34% hit rate. The route cache is described with Figure 14.34 and the group address cache with Figure 14.21.

## SNMP Variables

There is no standard SNMP MIB for multicast routing, but [McCloghrie and Farinacci 1994a] and [McCloghrie and Farinacci 1994b] describe some experimental MIBs for multicast routers.

---

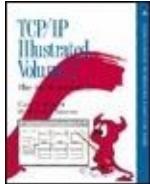
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

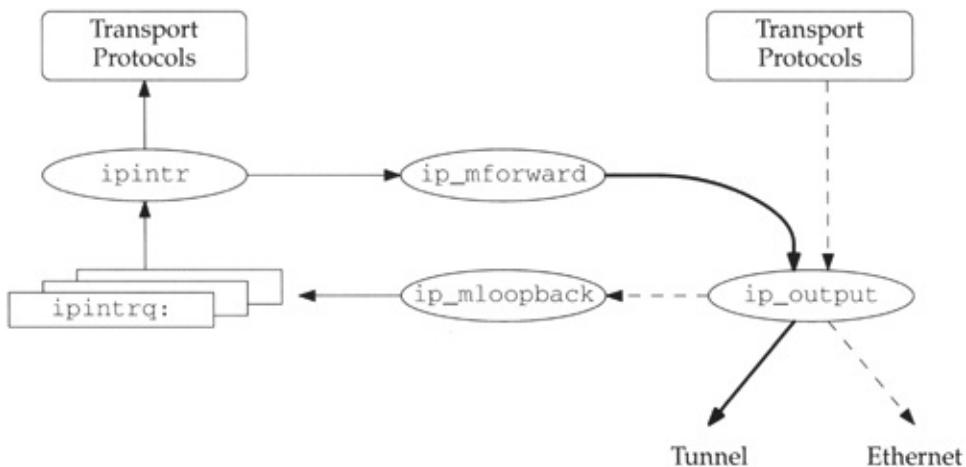
### 14.3 Multicast Output Processing Revisited

In [Section 12.15](#) we described how an interface is selected for an outgoing multicast datagram. We saw that `ip_output` is passed an explicit interface in the `ip_moptions` structure, or `ip_output` looks up the destination group in the routing tables and uses the interface returned in the route entry.

If, after selecting an outgoing interface, `ip_output` loops back the datagram, it is queued for input processing on the interface selected for *output* and is considered for forwarding when it is processed by `ipintr`. [Figure 14.6](#) illustrates

this process.

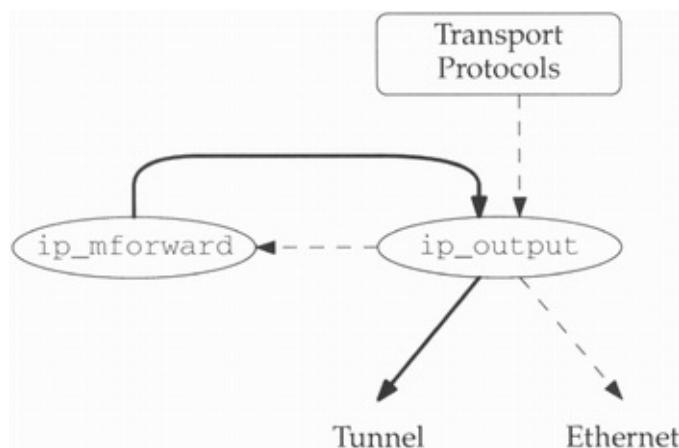
## Figure 14.6. Multicast output processing with loopback.



In Figure 14.6 the dashed arrows represent the original outgoing datagram, which in this example is multicast on a local Ethernet. The copy created by `ip_mloopback` is represented by the thin arrows; this copy is passed to the transport protocols for input. The third copy is created when `ip_mforward` decides to forward the datagram through another interface on the system. The thickest arrows in Figure 14.6 represents the third copy, which in this example is sent on a multicast tunnel.

If the datagram is *not* looped back, `ip_output` passes it directly to `ip_mforward`, where it is duplicated and also processed as if it were received on the interface that `ip_output` selected. This process is shown in [Figure 14.7](#).

**Figure 14.7. Multicast output processing with no loopback.**



Whenever `ip_mforward` calls `ip_output` to send a multicast datagram, it sets the `IP_FORWARDING` flag so that `ip_output` does not pass the datagram back to `ip_mforward`, which would create an infinite loop.

`ip_mloopback` was described with [Figure](#)

[12.42.](#) `ip_mforward` is described in [Section 14.8](#).

---

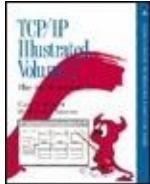
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.4 mrouted Daemon

Multicast routing is enabled and managed by a user-level process: the mrouted daemon, mrouted implements the router portion of the IGMP protocol and communicates with other multicast routers to implement multicast routing between networks. The routing algorithms are implemented in mrouted, but the multicast routing tables are maintained in the kernel, which forwards the datagrams.

In this text we describe only the kernel data structures and functions that support mroutedwe do not describe mrouted itself. We describe the Truncated Reverse Path Broadcast (TRPB) algorithm [Deering and

[Cheriton 1990](#)], used to select routes for multicast datagrams, and the Distance Vector Multicast Routing Protocol (DVMRP), used to convey information between multicast routers, in enough detail to make sense of the kernel multicast code.

RFC 1075 [[Waitzman, Partridge, and Deering 1988](#)] describes an old version of DVMRP. mrouted implements a newer version of DVMRP, which is not yet documented in an RFC. The best documentation for the current algorithm and protocol is the source code release for mrouted. [Appendix B](#) describes where the source code can be obtained.

The mrouted daemon communicates with the kernel by setting options on an IGMP socket ([Chapter 32](#)). The options are summarized in [Figure 14.8](#).

## **Figure 14.8. Multicast routing socket options.**

optname	optval type	Function	Description
DVMRP_INIT		ip_mrouter_init	mrouterd is starting
DVMRP_DONE		ip_mrouter_done	mrouterd is shutting down
DVMRP_ADD_VIF	struct vifctl vifi_t	add_vif	add virtual interface
DVMRP_DEL_VIF		del_vif	delete virtual interface
DVMRP_ADD_LGRP	struct lgrpctl	add_lgrp	add multicast group entry for an interface
DVMRP_DEL_LGRP	struct lgrpctl	del_lgrp	delete multicast group entry for an interface
DVMRP_ADD_MRT	struct mrtctl	add_mrt	add multicast route
DVMRP_DEL_MRT	struct in_addr	del_mrt	delete multicast route

The socket options shown in Figure 14.8 are passed to `rip_ctloutput` (Section 32.8) by the `setsockopt` system call. Figure 14.9 shows the portion of `rip_ctloutput` that handles the DVMRP\_xxx options.

**Figure 14.9. `rip_ctloutput` function:  
DVMRP\_xxx socket options.**

---

```

173     case DVMRP_INIT:
174     case DVMRP_DONE:
175     case DVMRP_ADD_VIF:
176     case DVMRP_DEL_VIF:
177     case DVMRP_ADD_LGRP:
178     case DVMRP_DEL_LGRP:
179     case DVMRP_ADD_MRT:
180     case DVMRP_DEL_MRT:
181         if (op == PRCO_SETOPT) {
182             error = ip_mrouter_cmd(optname, so, *m);
183             if (*m)
184                 (void) m_free(*m);
185         } else
186             error = EINVAL;
187         return (error);

```

---

173-187

When `setsockopt` is called, `op` equals `PRCO_SETOPT` and all the options are passed to the `ip_mrouter_cmd` function.

For the getsockopt system call, op equals PRCO\_GETOPT and EINVAL is returned for all the options.

[Figure 14.10](#) shows the ip\_mrouter\_cmd function.

## Figure 14.10. ip\_mrouter\_cmd function.

```
ip_mroute.c
84 int
85 ip_mrouter_cmd(cmd, so, m)
86 int      cmd;
87 struct socket *so;
88 struct mbuf *m;
89 {
90     int      error = 0;
91     if (cmd != DVMRP_INIT && so != ip_mrouter)
92         error = EACCES;
93     else
94         switch (cmd) {
95             case DVMRP_INIT:
96                 error = ip_mrouter_init(so);
97                 break;
98             case DVMRP_DONE:
99                 error = ip_mrouter_done();
100                break;
101            case DVMRP_ADD_VIF:
102                if (m == NULL || m->m_len < sizeof(struct vifctl))
103                    error = EINVAL;
104                else
105                    error = add_vif(mtod(m, struct vifctl *));
106                break;
107            case DVMRP_DEL_VIF:
108                if (m == NULL || m->m_len < sizeof(short))
109                    error = EINVAL;
110                else
111                    error = del_vif(mtod(m, vifi_t *));
112                break;
113        }
114    }
```

```

113     case DVMRP_ADD_LGRP:
114         if (m == NULL || m->m_len < sizeof(struct lgrpctl))
115             error = EINVAL;
116         else
117             error = add_lgrp(mtod(m, struct lgrpctl *));
118         break;
119
120     case DVMRP_DEL_LGRP:
121         if (m == NULL || m->m_len < sizeof(struct lgrpctl))
122             error = EINVAL;
123         else
124             error = del_lgrp(mtod(m, struct lgrpctl *));
125         break;
126
127     case DVMRP_ADD_MRT:
128         if (m == NULL || m->m_len < sizeof(struct mrtctl))
129             error = EINVAL;
130         else
131             error = add_mrt(mtod(m, struct mrtctl *));
132         break;
133
134     case DVMRP_DEL_MRT:
135         if (m == NULL || m->m_len < sizeof(struct in_addr))
136             error = EINVAL;
137         else
138             error = del_mrt(mtod(m, struct in_addr *));
139         break;
140     default:
141         error = EOPNOTSUPP;
142     }
141     return (error);
142 }
```

*ip\_mroute.c*

These "options" are more like commands, since they cause the kernel to update various data structures. We use the term *command* throughout the rest of this chapter to emphasize this fact.

**84-92**

The first command issued by mrouted must be DVMRP\_INIT. Subsequent commands must come from the same socket as the DVMRP\_INIT command. EACCES is returned when other commands

are issued on a different socket.

94-142

Each case in the switch checks to see if the right amount of data was included with the command and then calls the matching function. If the command is not recognized, EOPNOTSUPP is returned. Any error returned from the matching function is posted in error and returned at the end of the function.

[Figure 14.11](#) shows ip\_mrouter\_init, which is called when mrouted issues the DVMRP\_INIT command during initialization.

**Figure 14.11. ip\_mrouter\_init function:  
DVMRP\_INIT command.**

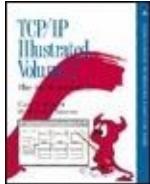
```
146 static int  
147 ip_mrouter_init(so)  
148 struct socket *so;  
149 {  
150     if (so->so_type != SOCK_RAW ||  
151         so->so_proto->pr_protocol != IPPROTO_IGMP)  
152         return (EOPNOTSUPP);  
153     if (ip_mrouter != NULL)  
154         return (EADDRINUSE);  
155     ip_mrouter = so;  
156     return (0);  
157 }
```

ip\_mroute.c

## 146-157

If the command is issued on something other than a raw IGMP socket, or if DVMRP\_INIT has already been set, EOPNOTSUPP or EADDRINUSE are returned respectively. A pointer to the socket on which the initialization command is issued is saved in the global ip\_mrouter. Subsequent commands must be issued on this socket. This prevents the concurrent operation of more than one instance of mrouted.

The remainder of the DVMRP\_xxx commands are described in the following sections.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

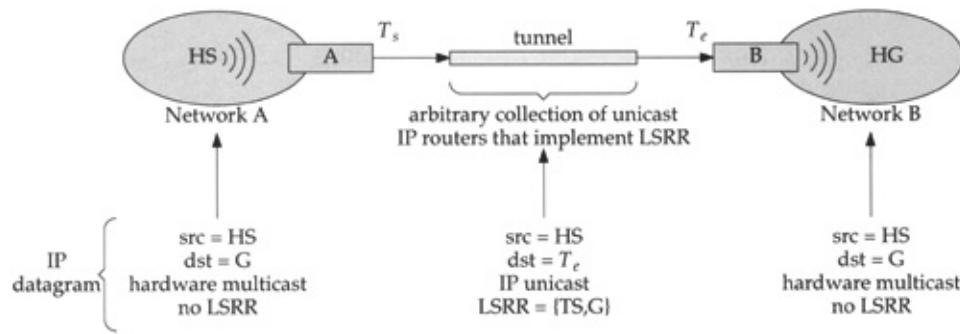
### 14.5 Virtual Interfaces

When operating as a multicast router, Net/3 accepts incoming multicast datagrams, duplicates them and forwards the copies through one or more interfaces. In this way, the datagram is forwarded to other multicast routers on the internet.

An outgoing interface can be a physical interface or it can be a multicast *tunnel*. Each end of the multicast tunnel is associated with a physical interface on a multicast router. Multicast tunnels allow two multicast routers to exchange multicast datagrams even when they are separated by routers that cannot forward multicast datagrams. [Figure 14.12](#) shows

two multicast routers connected by a multicast tunnel.

**Figure 14.12. A multicast tunnel.**



In Figure 14.12, the source host HS on network A is multicasting a datagram to group G. The only member of group G is on network B, which is connected to network A by a multicast tunnel. Router A receives the multicast (because multicast routers receive *all* multicasts), consults its multicast routing tables, and forwards the datagram through the multicast tunnel.

The tunnel starts on the *physical* interface on router A identified by the IP unicast address  $T_s$ . The tunnel ends on the *physical* interface on router B identified by the IP unicast address,  $T_e$ . The tunnel

itself is an arbitrarily complex collection of networks connected by IP unicast routers that implement the LSRR option. [Figure 14.13](#) shows how an IP LSRR option implements the multicast tunnel.

### **Figure 14.13. LSRR multicast tunnel options.**

System	IP header		Source route option		Description
	ip_src	ip_dst	offset	addresses	
HS	HS	G			on network A
$T_s$	HS	$T_e$	8	$T_s \bullet G$	on tunnel
$T_e$	HS	G	12	$T_s \text{ see text } \bullet$	after ip_dooptions on router B
$T_e$	HS	G			after ip_mforward on router B

The first line of [Figure 14.13](#) shows the datagram sent by HS as a multicast on network A. Router A receives the datagram because multicast routers receive all multicasts on their locally attached networks.

To send the datagram through the tunnel, router A inserts an LSRR option in the IP header. The second line shows the datagram as it leaves A on the tunnel. The first address in the LSRR option is the source address of the tunnel and the

second address is the destination group. The destination of the datagram is  $T_e$  the other end of the tunnel. The LSRR offset points to the *destination group*.

The tunneled datagram is forwarded through the internet until it reaches the other end of the tunnel on router B.

The third line of the figure shows the datagram after it is processed by ip\_dooptions on router B. Recall from [Chapter 9](#) that ip\_dooptions processes the LSRR option before the destination address of the datagram is examined by ipintr. Since the destination address of the datagram ( $T_e$ ) matches one of the interfaces on router B, ip\_dooptions copies the address identified by the option offset (G in this example) into the destination field of the IP header. In the option, G is replaced with the address returned by ip\_rtaddr, which normally selects the outgoing interface for the datagram based on the IP destination address (G in this case). This address is irrelevant, since ip\_mforward discards the entire option. Finally, ip\_dooptions advances the option

offset.

The fourth line in [Figure 14.13](#) shows the datagram after ipintr calls ip\_mforward, where the LSRR option is recognized and removed from the datagram header. The resulting datagram looks like the original multicast datagram and is processed by ip\_mforward, which in our example forwards it onto network B as a multicast datagram where it is received by HG.

Multicast tunnels constructed with LSRR options are obsolete. Since the March 1993 release of mrouted, tunnels have been constructed by prepending another IP header to the IP multicast datagram. The protocol in the new IP header is set to 4 to indicate that the contents of the packet is another IP packet. This value is documented in RFC 1700 as the "IP in IP" protocol. LSRR tunnels are supported in newer versions of mrouted for backward compatibility.

## Virtual Interface Table

For both physical interfaces and tunnel interfaces, the kernel maintains an entry in a *virtual interface* table, which contains information that is used only for multicasting. Each virtual interface is described by a vif structure (Figure 14.14). The global variable viftable is an array of these structures. An index to the table is stored in a vifi\_t variable, which is an unsigned short integer.

**Figure 14.14. vif structure.**

```
ip_mroute.h
105 struct vif {
106     u_char    v_flags;           /* VIFF_ flags */
107     u_char    v_threshold;      /* min ttl required to forward on vif */
108     struct in_addr v_lcl_addr; /* local interface address */
109     struct in_addr v_rmt_addr; /* remote address (tunnels only) */
110     struct ifnet *v_ifp;        /* pointer to interface */
111     struct in_addr *v_lcl_grps; /* list of local grps (phyints only) */
112     int       v_lcl_grps_max;   /* malloc'ed number of v_lcl_grps */
113     int       v_lcl_grps_n;     /* used number of v_lcl_grps */
114     u_long   v_cached_group;   /* last grp looked-up (phyints only) */
115     int       v_cached_result; /* last look-up result (phyints only) */
116 };
ip_mroute.h
```

105-110

The only flag defined for v\_flags is VIFF\_TUNNEL. When set, the interface is a tunnel to a remote multicast router. When not set, the interface is a physical interface on the local system. v\_threshold is the

multicast threshold, which we described in [Section 12.9](#). `v_lcl_addr` is the unicast IP address of the local interface associated with this virtual interface. `v_rmt_addr` is the unicast IP address of the remote end of an IP multicast tunnel. Either `v_lcl_addr` or `v_rmt_addr` is nonzero, but never both. For physical interfaces, `v_ifp` is nonnull and points to the ifnet structure of the local interface. For tunnels, `v_ifp` is null.

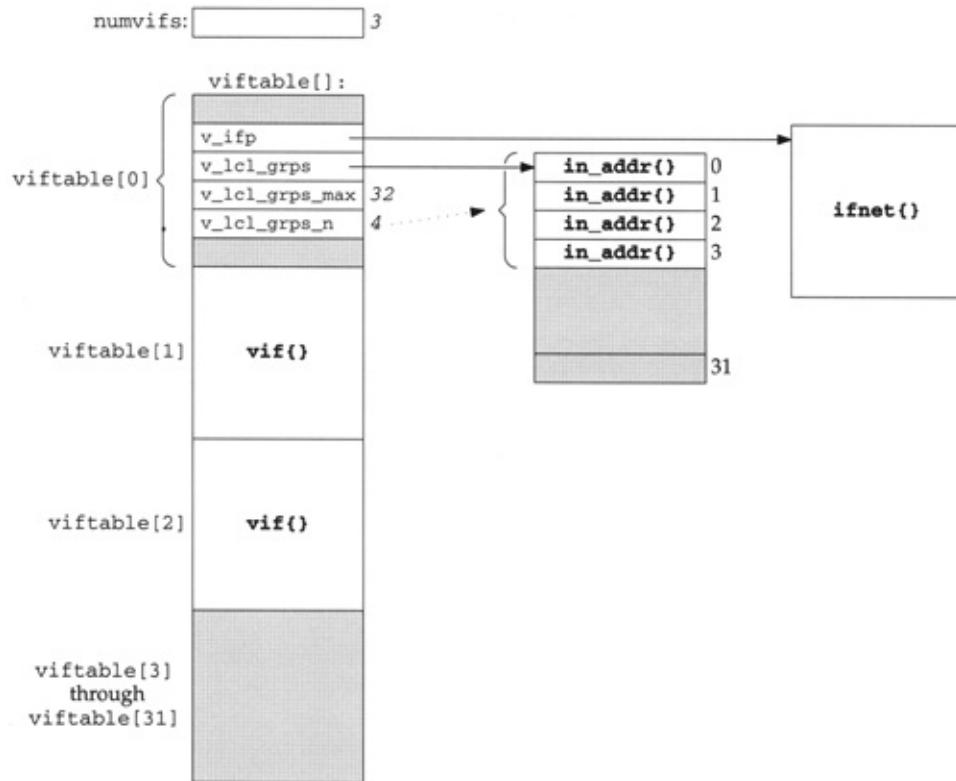
**111-116**

The list of groups with members on the attached interface is kept as an array of IP multicast group addresses pointed to by `v_lcl_grps`, which is always null for tunnels. The size of the array is in `v_lcl_grps_max`, and the number of entries that are used is in `v_lcl_grps_n`. The array grows as needed to accommodate the group membership list. `v_cached_group` and `v_cached_result` implement a one-entry cache, which contain the group and result of the previous lookup.

[Figure 14.15](#) illustrates the viftable, which has 32 (MAXVIFS) entries. `viftable[2]` is

the last entry in use, so numvifs is 3. The size of the table is fixed when the kernel is compiled. Several members of the vif structure in the first entry of the table are shown. v\_ifp points to an ifnet structure, v\_lcl\_grps points to an array of in\_addr structures. The array has 32 (v\_lcl\_grps\_max) entries, of which only 4 (v\_lcl\_grps\_n) are in use.

**Figure 14.15. viftable array.**



mouted maintains viftable through the

DVMRP\_ADD\_VIF and DVMRP\_DEL\_VIF commands. Normally all multicast-capable interfaces on the local system are added to the table when mrouted begins. Multicast tunnels are added when mrouted reads its configuration file, usually /etc/mrouted.conf. Commands in this file can also delete physical interfaces from the virtual interface table or change the multicast information associated with the interfaces.

A vifctl structure ([Figure 14.16](#)) is passed by mrouted to the kernel with the DVMRP\_ADD\_VIF command. It instructs the kernel to add an interface to the table of virtual interfaces.

## Figure 14.16. vifctl structure.

```
76 struct vifctl {
77     vifi_t vifc_vifi;           /* the index of the vif to be added */
78     u_char vifc_flags;          /* VIFF_ flags (Figure 14.14) */
79     u_char vifc_threshold;      /* min ttl required to forward on vif */
80     struct in_addr vifc_lcl_addr; /* local interface address */
81     struct in_addr vifc_rmt_addr; /* remote address (tunnels only) */
82 };
```

vifc\_vifi identifies the index of the virtual interface within viftable. The remaining four members, vifc\_flags, vifc\_threshold, vifc\_lcl\_addr, and vifc\_rmt\_addr, are copied into the vif structure by the add\_vif function.

## **add\_vif Function**

Figure 14.17 shows the add\_vif function.

**Figure 14.17. add\_vif function:  
DVMRP\_ADD\_VIF command.**

---

```

202 static int
203 add_vif(vifcp)
204 struct vifctl *vifcp;
205 {
206     struct vif *vifp = viftable + vifcp->vifc_vifi;
207     struct ifaddr *ifa;
208     struct ifnet *ifp;
209     struct ifreq ifr;
210     int error, s;
211     static struct sockaddr_in sin =
212         (sizeof(sin), AF_INET);
213
214     if (vifcp->vifc_vifi >= MAXVIFS)
215         return (EINVAL);
216     if (vifp->v_lcl_addr.s_addr != 0)
217         return (EADDRINUSE);
218
219     /* Find the interface with an address in AF_INET family */
220     sin.sin_addr = vifcp->vifc_lcl_addr;
221     ifa = ifa_ifwithaddr((struct sockaddr *) &sin);
222     if (ifa == 0)
223         return (EADDRNOTAVAIL);
224
225     s = splnet();
226
227     if (vifcp->vifc_flags & VIFF_TUNNEL)
228         vifp->v_rmt_addr = vifcp->vifc_rmt_addr;
229     else {
230         /* Make sure the interface supports multicast */
231         ifp = ifa->ifa_ifp;
232         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
233             splx(s);
234             return (EOPNOTSUPP);
235         }
236         /*
237          * Enable promiscuous reception of all IP multicasts
238          * from the interface.
239          */
240         satosin(&ifr.ifr_addr)->sin_family = AF_INET;
241         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
242         error = (*ifp->if_ioctl) (ifp, SIOCADDMULTI, (caddr_t) &ifr);
243         if (error) {
244             splx(s);
245             return (error);
246         }
247     }
248     vifp->v_flags = vifcp->vifc_flags;
249     vifp->v_threshold = vifcp->vifc_threshold;
250     vifp->v_lcl_addr = vifcp->vifc_lcl_addr;
251     vifp->v_ifp = ifa->ifa_ifp;
252
253     /* Adjust numvifs up if the wifi is higher than numvifs */
254     if (numvifs <= vifcp->vifc_vifi)
255         numvifs = vifcp->vifc_vifi + 1;
256
257     splx(s);
258     return (0);
259 }

```

---

ip\_mroute.c

## Validate index

202-216

If the table index specified by mrouted in vifc\_vifi is too large, or the table entry is already in use, EINVAL or EADDRINUSE is returned respectively.

## Locate physical interface

217-221

ifa\_ifwithaddr takes the unicast IP address in vifc\_lcl\_addr and returns a pointer to the associated ifnet structure. This identifies the physical interface to be used for this virtual interface. If there is no matching interface, EADDRNOTAVAIL is returned.

## Configure tunnel interface

222-224

For a tunnel, the remote end of the tunnel is copied from the vifctl structure to the vif structure in the interface table.

## Configure physical interface

225-243

For a physical interface, the link-level driver must support multicasting. The SIOCADDMULTI command used with INADDR\_ANY configures the interface to begin receiving *all* IP multicast datagrams ([Figure 12.32](#)) because it is a multicast router. Incoming datagrams are forwarded when ipintr passes them to ip\_mforward.

## Save multicast information

244-253

The remaining interface information is copied from the vifctl structure to the vif structure. If necessary, numvifs is updated to record the number of virtual interfaces in use.

## del\_vif Function

The function del\_vif, shown in [Figure 14.18](#), deletes entries from the virtual interface table. It is called when mrouted sets the DVMRP\_DEL\_VIF command.

## Figure 14.18. del\_vif function: DVMRP\_DEL\_VIF command.

```
257 static int  
258 del_vif(vifip)  
259 vifi_t *vifip;  
260 {  
261     struct vif *vifp = viftable + *vifip;  
262     struct ifnet *ifp;  
263     int i, s;  
264     struct ifreq ifr;  
265     if (*vifip >= numvifs)  
266         return (EINVAL);  
267     if (vifp->v_lcl_addr.s_addr == 0)  
268         return (EADDRNOTAVAIL);  
269     s = splnet();  
270     if (!(vifp->v_flags & VIFF_TUNNEL)) {  
271         if (vifp->v_lcl_grps)  
272             free(vifp->v_lcl_grps, M_MRTABLE);  
273         satosin(&ifr.ifr_addr)->sin_family = AF_INET;  
274         satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;  
275         ifp = vifp->v_ifp;  
276         (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);  
277     }  
278     bzero((caddr_t) vifp, sizeof(*vifp));  
279     /* Adjust numvifs down */  
280     for (i = numvifs - 1; i >= 0; i--)  
281         if (viftable[i].v_lcl_addr.s_addr != 0)  
282             break;  
283     numvifs = i + 1;  
284     splx(s);  
285     return (0);  
286 }
```

## Validate index

257-268

If the index passed to del\_vif is greater than the largest index in use or it references an entry that is not in use, EINVAL or EADDRNOTAVAIL is returned

respectively.

## Delete interface

269-278

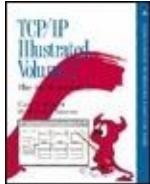
For a physical interface, the local group table is released, and the reception of all multicast datagrams is disabled by SIOCDELMULTI. The entry in viftable is cleared by bzero.

## Adjust interface count

279-286

The for loop searches for the first active entry in the table starting at the largest previously active entry and working back toward the first entry. For unused entries, the s\_addr member of v\_lcl\_addr (an in\_addr structure) is 0. numvifs is updated accordingly and the function returns.

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.6 IGMP Revisited

Chapter 13 focused on the host part of the IGMP protocol. mrouted implements the router portion of this protocol. For every physical interface, mrouted must keep track of which multicast groups have members on the attached network. mrouted multicasts an `IGMP_HOST_MEMBERSHIP_QUERY` datagram every 120 seconds and compiles the resulting `IGMP_HOST_MEMBERSHIP_REPORT` datagrams into a membership array associated with each network. This array is *not* the same as the membership list we described in Chapter 13.

From the information collected, mrouted constructs the multicast routing tables. The list of groups is also used to suppress multicasts to areas of the multicast internet that do not have members of the destination group.

The membership array is maintained only for physical interfaces. Tunnels are point-to-point interfaces to another multicast router, so no group membership information is needed.

We saw in [Figure 14.14](#) that v\_lcl\_grps points to an array of IP multicast groups. mrouted maintains this list with the DVMRP\_ADD\_LGRP and DVMRP\_DEL\_LGRP commands. An Igrplctl ([Figure 14.19](#)) structure is passed with both commands.

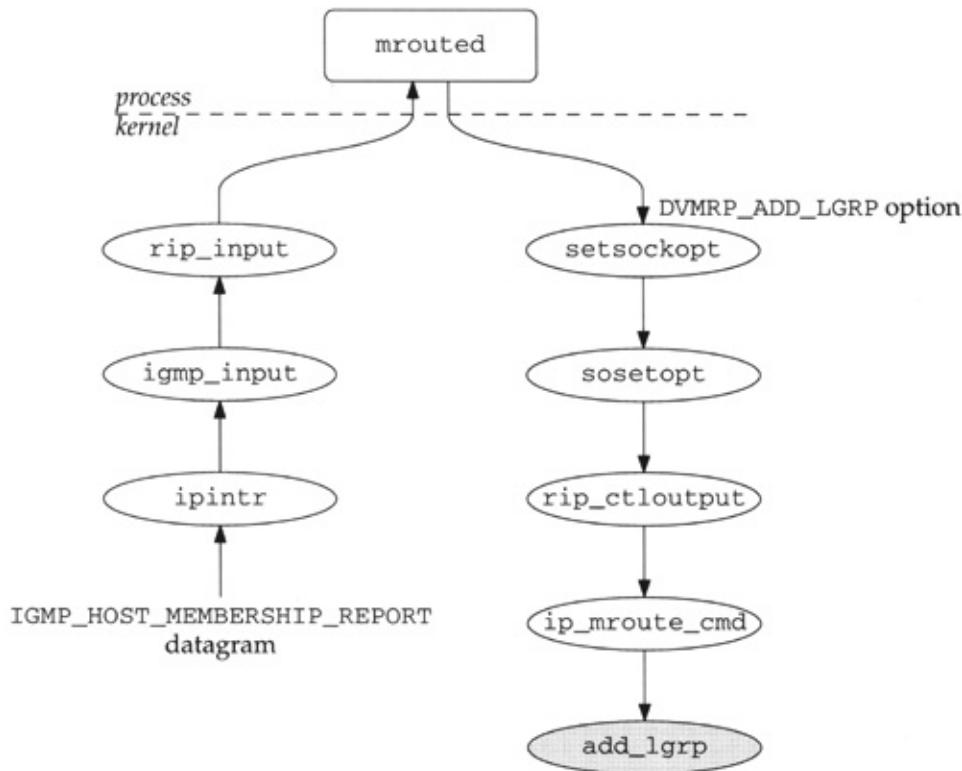
## Figure 14.19. Igrplctl structure.

```
87 struct lgrplctl {  
88     vifi_t    lgc_vifi;  
89     struct in_addr lgc_gaddr;  
90 };
```

The {interface, group} pair is identified by lgc\_vifi and lgc\_gaddr. The interface index (lgc\_vifi, an unsigned short) identifies a *virtual* interface, not a physical interface.

When an  
IGMP\_HOST\_MEMBERSHIP\_REPORT  
datagram is received, the functions shown  
in Figure 14.20 are called.

**Figure 14.20. IGMP report processing.**



## add\_lgrp Function

mrouted examines the source address of an incoming IGMP report to determine which subnet and therefore which interface the report arrived on. Based on this information, mrouted sets the DVMRP\_ADD\_LGRP command for the interface to update the membership table in the kernel. This information is also fed into the multicast routing algorithm to update the routing tables. [Figure 14.21](#) shows the add\_lgrp function.

**Figure 14.21. add\_lgrp function: process DVMRP\_ADD\_LGRP command.**

```

291 static int
292 add_lgrp(gcp)
293 struct lgrpctl *gcp;
294 {
295     struct vif *vifp;
296     int     s;
297
298     if (gcp->lgc_vifi >= numvifs)
299         return (EINVAL);
300
301     vifp = viftable + gcp->lgc_vifi;
302     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
303         return (EADDRNOTAVAIL);
304
305     /* If not enough space in existing list, allocate a larger one */
306     s = splnet();
307     if (vifp->v_lcl_grps_n + 1 >= vifp->v_lcl_grps_max) {
308         int     num;
309         struct in_addr *ip;
310
311         num = vifp->v_lcl_grps_max;
312         if (num <= 0)
313             num = 32;           /* initial number */
314         else
315             num += num;        /* double last number */
316         ip = (struct in_addr *) malloc(num * sizeof(*ip),
317                                         M_MRTABLE, M_NOWAIT);
318         if (ip == NULL) {
319             splx(s);
320             return (ENOBUFS);
321         }
322         bzero((caddr_t) ip, num * sizeof(*ip));    /* XXX paranoid */
323         bcopy((caddr_t) vifp->v_lcl_grps, (caddr_t) ip,
324               vifp->v_lcl_grps_n * sizeof(*ip));
325
326         vifp->v_lcl_grps_max = num;
327         if (vifp->v_lcl_grps)
328             free(vifp->v_lcl_grps, M_MRTABLE);
329         vifp->v_lcl_grps = ip;
330
331         splx(s);
332     }
333     vifp->v_lcl_grps[vifp->v_lcl_grps_n++] = gcp->lgc_gaddr;
334
335     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
336         vifp->v_cached_result = 1;
337
338     splx(s);
339     return (0);
340 }

```

ip\_mroute.c

## Validate add request

291-301

If the request identifies an invalid interface, EINVAL is returned. If the interface is not in use or is a tunnel,

EADDRNOTAVAIL is returned.

## If needed, expand group array

302-326

If the new group won't fit in the current group array, a new array is allocated. The first time add\_lgrp is called for an interface, an array is allocated to hold 32 groups.

Each time the array fills, add\_lgrp allocates a new array of twice the previous size. The new array is allocated by malloc, cleared by bzero, and filled by copying the old array into the new one with bcopy. The maximum number of entries, v\_lcl\_grps\_max, is updated, the old array (if any) is released, and the new array is attached to the vif entry with v\_lcl\_grps.

The "paranoid" comment points out there is no guarantee that the memory allocated by malloc contains all 0s.

## Add new group

327-332

The new group is copied into the next available entry and if the cache already contains the new group, the cache is marked as valid.

The lookup cache contains an address, v\_cached\_group, and a cached lookup result, v\_cached\_result. The grplst\_member function always consults the cache before searching the membership array. If the given group matches v\_cached\_group, the cached result is returned; otherwise the membership array is searched.

## del\_lgrp Function

Group information is expired for each interface when no membership report has been received for the group within 270 seconds. mrouted maintains the appropriate timers and issues the DVMRP\_DEL\_LGRP command when the information expires. [Figure 14.22](#) shows del\_lgrp.

## Figure 14.22. del\_lgrp function: process DVMRP\_DEL\_LGRP command.

```
ip_mroute.c
337 static int
338 del_lgrp(gcp)
339 struct lgrpctl *gcp;
340 {
341     struct vif *vifp;
342     int i, error, s;
343     if (gcp->lgc_vifi >= numvifs)
344         return (EINVAL);
345     vifp = viftable + gcp->lgc_vifi;
346     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
347         return (EADDRNOTAVAIL);
348     s = splnet();
349     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
350         vifp->v_cached_result = 0;
351     error = EADDRNOTAVAIL;
352     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
353         if (same(&gcp->lgc_gaddr, &vifp->v_lcl_grps[i])) {
354             error = 0;
355             vifp->v_lcl_grps_n--;
356             bcopy((caddr_t) & vifp->v_lcl_grps[i + 1],
357                   (caddr_t) & vifp->v_lcl_grps[i],
358                   (vifp->v_lcl_grps_n - i) * sizeof(struct in_addr));
359             error = 0;
360             break;
361         }
362     splx(s);
363     return (error);
364 }
```

ip\_mroute.c

## Validate interface index

337-347

If the request identifies an invalid interface, EINVAL is returned. If the interface is not in use or is a tunnel, EADDRNOTAVAIL is returned.

## **Update lookup cache**

**348-350**

If the group to be deleted is in the cache, the lookup result is set to 0 (false).

## **Delete group**

**351-364**

EADDRNOTAVAIL is posted in error in case the group is not found in the membership list. The for loop searches the membership array associated with the interface. If same (a macro that uses bcmp to compare the two addresses) is true, error is cleared and the group count is decremented. bcopy shifts the subsequent array entries down to delete the group and del\_lgrp breaks out of the loop.

If the loop completes without finding a match, EADDRNOTAVAIL is returned; otherwise 0 is returned.

## **grplst\_member Function**

During multicast forwarding, the membership array is consulted to avoid sending datagrams on a network when no member of the destination group is present. `grplst_member`, shown in [Figure 14.23](#), searches the list looking for the given group address.

## Figure 14.23. `grplst_member` function.

```
368 static int          . ip_mroute.c
369 grplst_member(vifp, gaddr)
370 struct vif *vifp;
371 struct in_addr gaddr;
372 {
373     int     i, s;
374     u_long  addr;
375     mrtstat.mrts_grp_lookups++;
376     addr = gaddr.s_addr;
377     if (addr == vifp->v_cached_group)
378         return (vifp->v_cached_result);
379     mrtstat.mrts_grp_misses++;
380     for (i = 0; i < vifp->v_lcl_grps_n; ++i)
381         if (addr == vifp->v_lcl_grps[i].s_addr) {
382             s = splnet();
383             vifp->v_cached_group = addr;
384             vifp->v_cached_result = 1;
385             splx(s);
386             return (1);
387         }
388         s = splnet();
389         vifp->v_cached_group = addr;
390         vifp->v_cached_result = 0;
391     splx(s);
392     return (0);
393 }
```

## Check the cache

368-379

If the requested group is located in the cache, the cached result is returned and the membership array is not searched.

## Search the membership array

380-393

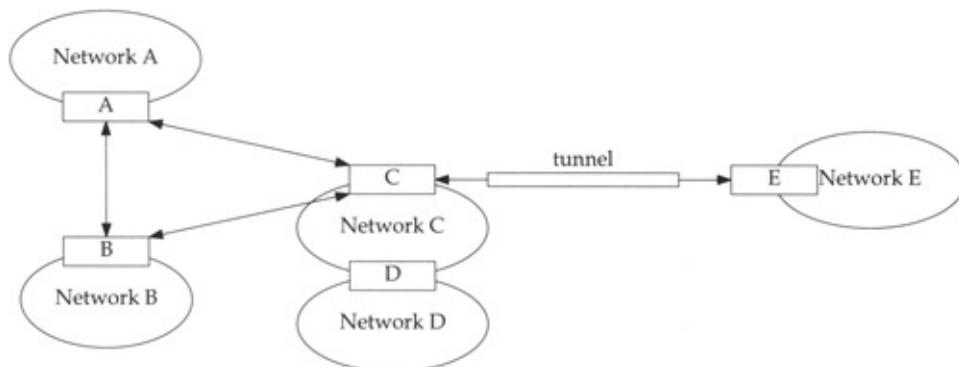
A linear search determines if the group is in the array. If it is found, the cache is updated to record the match and one is returned. If it is not found, the cache is updated to record the miss and 0 is returned.

## Chapter 14. IP Multicast Routing

### 14.7 Multicast Routing

As we mentioned at the start of this chapter, while presenting the TRPB algorithm implemented by the kernel, we need to provide a general overview of the mechanics of the multicast routing table and the multicast routing algorithm used by the kernel. [Figure 14.24](#) shows the sample multicast routing topology used to illustrate the algorithms.

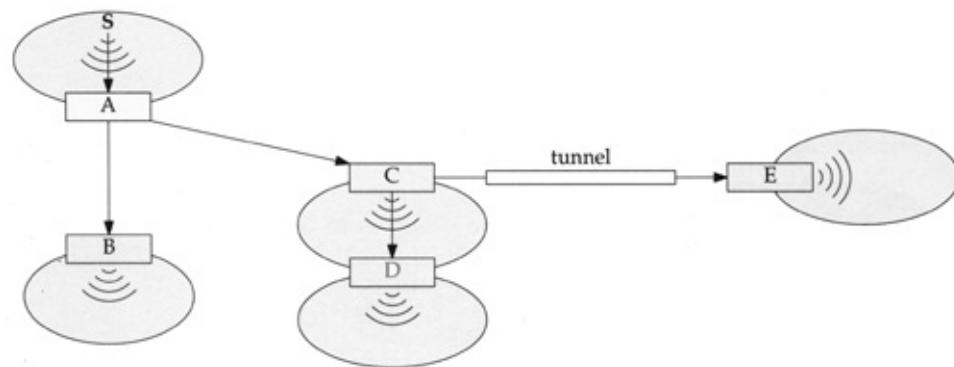
**Figure 14.24. Sample multicast routing topology**



In Figure 14.24, routers are shown as boxes and multicast networks attached to the routers. Router A can multicast on network D and C. Router C can multicast on network C, to routers A and B through point-to-point links, and to E through a multicast tunnel.

The simplest approach to multicast routing is to use the internet topology that forms a *spanning tree*. A router forwards multicasts along the spanning tree, even if it receives the datagram. Figure 14.25 shows one such a sample network, where host S on network A is the source of a multicast datagram.

**Figure 14.25. Spanning tree for network A**



For a discussion of spanning trees, see [Taner [Perlman 1992]].

We constructed the tree based on the shortest path from every network back to the source in network A.

link between routers B and C is omitted to form a spanning tree. The arrows between the source and router A, and between router D and the destination on network C, emphasize that the multicast network is a spanning tree.

If the same spanning tree were used to forward a datagram from network C, the datagram would be forwarded along a path longer than needed to get to a recipient on network B. The RPB protocol, described in RFC 1075, computes a separate spanning tree for each potential source network to avoid this problem. Each host on a network contains a network number and subnet mask for its network. A single route applies to any host within the source network.

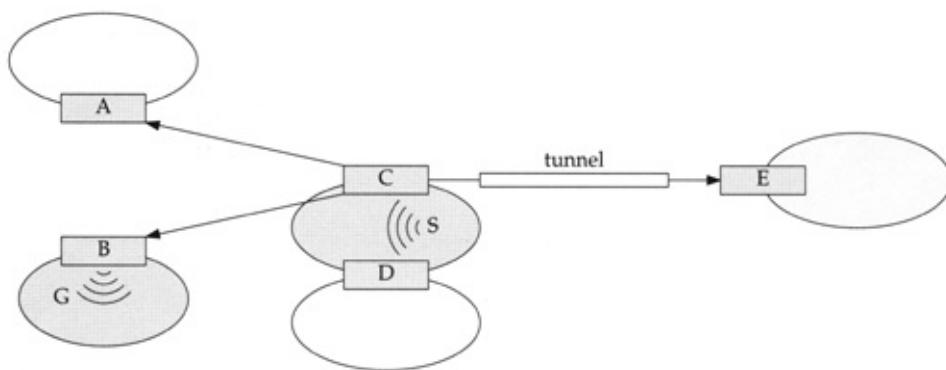
Because each spanning tree is constructed to provide a reverse path to the source of the datagram, any router that receives every multicast datagram, this process is called *reverse path broadcasting* or RPB.

The RPB protocol has no knowledge of multicast groups, so many datagrams are unnecessarily forwarded. A router may have no members in the destination group. If, in the process of computing the spanning trees, the routing algorithm finds that some networks are *leafs* and is aware of the group membership of the network, then routers attached to leaf networks do not forward datagrams onto the network when the routers are not part of the destination group present. This is called *true reverse path broadcasting* (TRPB), and is implemented by version 2 of the RPB protocol with the help of IGMP to keep track of member groups.

networks.

Figure 14.26 shows TRPB applied to a multicast on network C and with a member of the destination network B.

**Figure 14.26. TRPB routing for network C**



We'll use Figure 14.26 to illustrate the terms used in a multicast routing table. In this example, the shared routers receive a copy of the multicast datagram sent by the source on network C. The link between A and B is part of the spanning tree and C does not have a link to D, so the traffic sent by the source is received directly by C and forwarded to B.

In this figure, networks A, B, D, and E are leaf networks. Network C receives the multicast and forwards it through its routers attached to routers A, B, and E even though some of the traffic is wasted effort. This is a major weakness of the TRPB protocol.

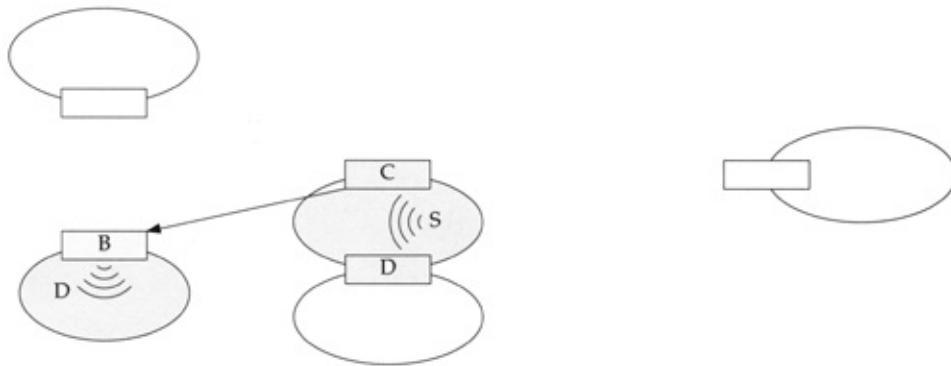
The interface associated with network C on router C is the *parent* because it is the interface on which router C receives multicasts originating from network C. The interfaces connecting router C to routers A, B, and E, are *child* interfaces. The point-to-point interface is the parent for the two networks, and the interface for network A is a child. Interfaces can act as a parent or as a child relative to the source of the multicast datagram. Multicast datagrams are forwarded only to the child interfaces, and never to the parent interface.

Continuing with the example, networks A, D, and E are leaf networks without members of the destination group, so the spanning tree is truncated at the interface connecting router C to network B. The datagram is not forwarded onto these networks because there is no member of the destination group on the network. To implement the reverse path forwarding algorithm, each multicast router that receives a datagram checks the group table associated with every virtual interface in its routing table.

The final refinement to the multicast routing algorithm is called *reverse path multicasting* (RPM). The goal of RPM is to build a spanning tree and avoid sending datagrams along branches of the tree that do not contain a member of the destination group. In [Figure 14.26](#), RPM would prevent router C from sending datagrams to A and E, since there is no member of the destination group on those branches of the tree. Version 3.3 of mroute

[Figure 14.27](#) shows our example network, but the routers and networks reached when the datagrams are shaded.

**Figure 14.27. RPM routing for network 1**



To compute the routing tables corresponding to what we described, the multicast routers communicate with other multicast routers to discover the multicast interface location of multicast group members. In Net/3, this communication is done via DVMRP. DVMRP messages are transported in IP datagrams and are sent to the multicast group reserved for DVMRP communication ([Figure 12.39](#)).

In [Figure 12.39](#), we saw that incoming IGMP packets accepted by a multicast router. They are passed to rip\_input, and then read by mrouted on a raw socket. mrouted sends DVMRP messages to other multicast routers on the same raw socket.

For more information about RPB, TRPB, RPM, and the messages that are needed to implement these [Deering and Cheriton 1990] and the source code, see the section on mroute.

There are other multicast routing protocols in use. Proteon routers implement the MOSPF protocol [RFC 1584] [Moy 1994]. PIM (Protocol Independent Multicast) is implemented by Cisco routers, starting with Release 10.3 operating software. PIM is described in [Deering and Cheriton 1990].

## Multicast Routing Table

We can now describe the implementation of the tables in Net/3. The kernel's multicast routing table is implemented as a hash table with 64 entries (MRTHASHSIZ). This is a global array mrttable, and each entry points to a linked list of mrt structures, shown in Figure 14.28.

Figure 14.28. mrt structure

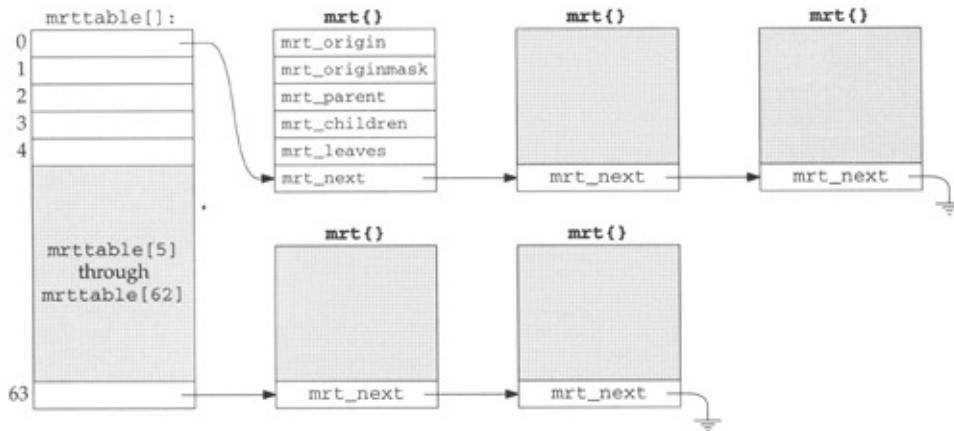
```
ip_mroute.h
120 struct mrt {
121     struct in_addr mrt_origin; /* subnet origin of multicasts */
122     struct in_addr mrt_originmask; /* subnet mask for origin */
123     vifi_t mrt_parent; /* incoming vif */
124     vifbitmap_t mrt_children; /* outgoing children vifs */
125     vifbitmap_t mrt_leaves; /* subset of outgoing children vifs */
126     struct mrt *mrt_next; /* forward link */
127 };
ip_mroute.h
```

120-127

mrtc\_origin and mrtc\_originmask identify an er  
mrtc\_parent is the index of the virtual interface  
multicast datagrams from the origin are expect  
interfaces are identified within mrtc\_children, w  
Outgoing interfaces that are also leaves in the i  
are identified in mrtc\_leaves, which is also a bit  
member, mrt\_next, implements a linked list in  
hash to the same array entry.

Figure 14.29 shows the organization of the mul  
Each mrt structure is placed in the hash chain t  
return value from the nethash function shown i

### Figure 14.29. Multicast routing



The multicast routing table maintained by the k  
the routing table maintained within mrouted an

information to support multicast forwarding with updates to the kernel table are sent with the D command, which includes the mrtctl structure shown in Figure 14.30.

**Figure 14.30. mrtctl structure**

```
95 struct mrtctl { ip_mroute.h
96     struct in_addr mrtc_origin; /* subnet origin of multicasts */
97     struct in_addr mrtc_originmask; /* subnet mask for origin */
98     vifi_t mrtc_parent; /* incoming vif */
99     vifbitmap_t mrtc_children; /* outgoing children vifs */
100    vifbitmap_t mrtc_leaves; /* subset of outgoing children vifs */
101 };
ip_mroute.h
```

95-101

The five members of the mrtctl structure carry have already described (Figure 14.28) between kernel.

The multicast routing table is keyed by the source IP address of the multicast datagram. nethash (Figure 14.31) implements the algorithm used for the table. It accepts the source IP address and returns a value between 0 and 63 (MRTHASHS)

**Figure 14.31. nethash function**

```
398 static u_long  
399 nethash(in)  
400 struct in_addr in;  
401 {  
402     u_long n;  
403     n = in_netof(in);  
404     while ((n & 0xff) == 0)  
405         n >>= 8;  
406     return (MRTHASHMOD(n));  
407 }
```

ip\_mroute.c

ip\_mroute.c

## 398-407

in\_netof returns in with the host portion set to the class A, B, or C network of the sending host shifted to the right until the low-order 8 bits are zero. MRTHASHMOD is

```
#define MRTHASHMOD (h)      ((h) & 0x000000ff)
```

The low-order 8 bits are logically ANDed with 6 low-order 6 bits, which is an integer in the range 0-63.

Doing two function calls (nethash and in\_netof) to get the hash value is an expensive algorithm to compute the 6-bit address.

## del\_mrt Function

The mrouted daemon adds and deletes entries in the multicast routing table through the DVMRP\_ADD and DVMRP\_DEL\_MRT commands. Figure 14.32 shows the del\_mrt function.

## Figure 14.32. del\_mrt function: process D command.

```
ip_mroute.c
451 static int
452 del_mrt(origin)
453 struct in_addr *origin;
454 {
455     struct mrt *rt, *prev_rt;
456     u_long hash = nethash(*origin);
457     int s;
458     for (prev_rt = rt = mrtable[hash]; rt; prev_rt = rt, rt = rt->mrt_next)
459         if (origin->s_addr == rt->mrt_origin.s_addr)
460             break;
461     if (!rt)
462         return (ESRCH);
463     s = splnet();
464     if (rt == cached_mrt)
465         cached_mrt = NULL;
466     if (prev_rt == rt)
467         mrtable[hash] = rt->mrt_next;
468     else
469         prev_rt->mrt_next = rt->mrt_next;
470     free(rt, M_MRTABLE);
471     splx(s);
472     return (0);
473 }
```

ip\_mroute.c

## Find route entry

451-462

The for loop starts at the entry identified by ha declaration from nethash). If the entry is not found returned.

## Delete route entry

463-473

If the entry was stored in the cache, the cache entry is unlinked from the hash chain and release is needed to handle the special case when the item is at the front of the list.

## **add\_mrt Function**

The add\_mrt function is shown in [Figure 14.33](#).

**Figure 14.33. add\_mrt function: process D command.**

---

```

411 static int
412 add_mrt(mrtcp)
413 struct mrtctl *mrtcp;
414 {
415     struct mrt *rt;
416     u_long hash;
417     int s;
418
419     if (rt = mrtfind(mrtcp->mrtc_origin)) {
420         /* Just update the route */
421         s = splnet();
422         rt->mrt_parent = mrtcp->mrtc_parent;
423         VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
424         VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
425         splx(s);
426         return (0);
427     }
428     s = splnet();
429
430     rt = (struct mrt *) malloc(sizeof(*rt), M_MRTABLE, M_NOWAIT);
431     if (rt == NULL) {
432         splx(s);
433         return (ENOBUFS);
434     }
435     /*
436      * insert new entry at head of hash chain
437      */
438     rt->mrt_origin = mrtcp->mrtc_origin;
439     rt->mrt_originmask = mrtcp->mrtc_originmask;
440     rt->mrt_parent = mrtcp->mrtc_parent;
441     VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
442     VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
443     /* link into table */
444     hash = nethash(mrtcp->mrtc_origin);
445     rt->mrt_next = mrttable[hash];
446     mrttable[hash] = rt;
447 }
```

---

ip\_mroute.c

## Update existing route

**411-427**

If the requested route is already in the routing information is copied into the route and add\_m

## Allocate new route

**428-447**

An mrt structure is constructed in a newly allocated memory from mrtctl structure passed with the information. The hash index is computed from mrtc\_origin, and the mrt is inserted as the first entry on the hash chain.

## mrtfind Function

The multicast routing table is searched with the mrtfind function. The source of the datagram is passed to mrtfind, and it returns a pointer to the matching mrt structure, or a null pointer if no match.

Figure 14.34. mrtfind function

```
477 static struct mrt *
478 mrtfind(origin)
479 struct in_addr origin;
480 {
481     struct mrt *rt;
482     u_int    hash;
483     int      s;
484     mrtstat.mrts_mrt_lookups++;
485     if (cached_mrt != NULL &&
486         (origin.s_addr & cached_originmask) == cached_origin)
487         return (cached_mrt);
488     mrtstat.mrts_mrt_misses++;
489     hash = nethash(origin);
490     for (rt = mrttable[hash]; rt; rt = rt->mrt_next)
491         if ((origin.s_addr & rt->mrt_originmask.s_addr) ==
492             rt->mrt_origin.s_addr) {
493             s = splnet();
494             cached_mrt = rt;
495             cached_origin = rt->mrt_origin.s_addr;
496             cached_originmask = rt->mrt_originmask.s_addr;
497             splx(s);
498             return (rt);
499         }
500     return (NULL);
501 }
```

ip\_mroute.c

## Check route lookup cache

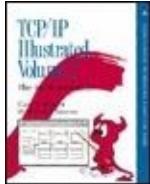
477-488

The given source IP address (origin) is logically ORed with the origin mask in the cache. If the result matches a cached entry, the cached entry is returned.

## Check the hash table

489-501

nethash returns the hash index for the route entry. It searches the hash chain for a matching route. If a match is found, the cache is updated and a pointer to the entry is returned. If a match is not found, a null pointer is returned.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.8 Multicast Forwarding: `ip_mforward` Function

Multicast forwarding is implemented entirely in the kernel. We saw in [Figure 12.39](#) that `ipintr` passes incoming multicast datagrams to `ip_mforward` when `ip_mrouter` is nonnull, that is, when `mouted` is running.

We also saw in [Figure 12.40](#) that `ip_output` can pass multicast datagrams that originate on the local host to `ip_mforward` to be routed to interfaces other than the one interface selected by `ip_output`.

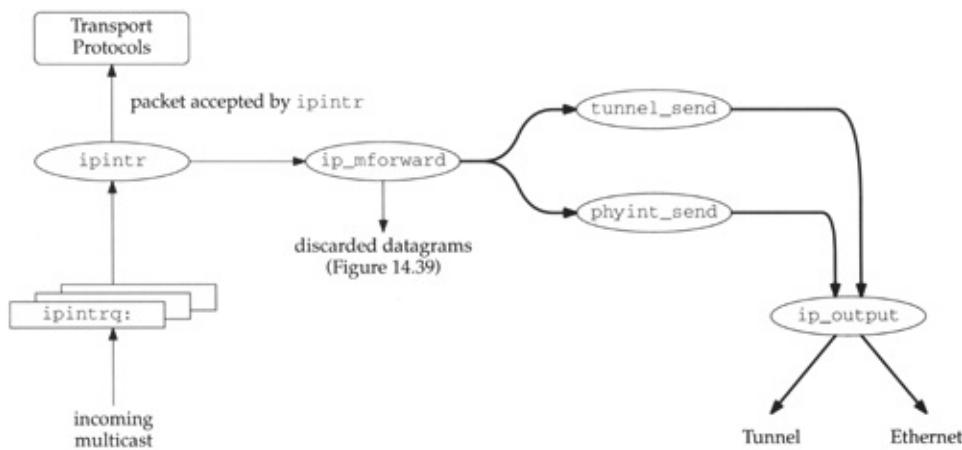
Unlike unicast forwarding, each time a multicast datagram is forwarded to an

interface, a copy is made. For example, if the local host is acting as a multicast router and is connected to three different networks, multicast datagrams originating on the system are duplicated and queued for *output* on all three interfaces.

Additionally, the datagram may be duplicated and queued for *input* if the multicast loopback flag was set by the application or if any of the outgoing interfaces receive their own transmissions.

[Figure 14.35](#) shows a multicast datagram arriving on a physical interface.

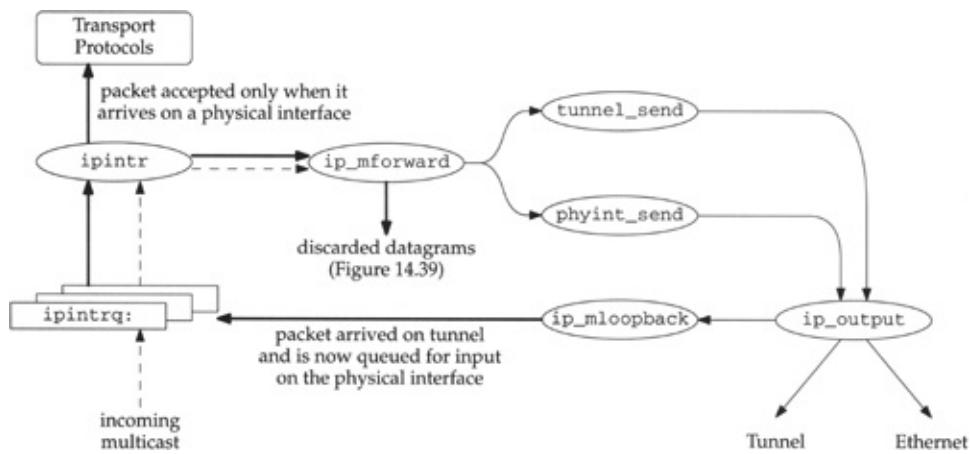
### Figure 14.35. Multicast datagram arriving on physical interface.



In Figure 14.35, the interface on which the datagram arrived is a member of the destination group, so the datagram is passed to the transport protocols for input processing. The datagram is also passed to ip\_mforward, where it is duplicated and forwarded to a physical interface and to a tunnel (the thick arrows), both of which must be different from the receiving interface.

Figure 14.36 shows a multicast datagram arriving on a tunnel.

### Figure 14.36. Multicast datagram arriving on a multicast tunnel.



In Figure 14.36, the datagram arriving on

a physical interface associated with the local end of the tunnel is represented by the dashed arrows. It is passed to `ip_mforward`, which as we'll see in [Figure 14.37](#) returns a nonzero value because the packet arrived on a tunnel. This causes `ipintr` to not pass the packet to the transport protocols.

**Figure 14.37. `ip_mforward` function:  
tunnel arrival.**

```

516 int
517 ip_mforward(m, ifp)
518 struct mbuf *m;
519 struct ifnet *ifp;
520 {
521     struct ip *ip = mtod(m, struct ip *);
522     struct mrt *rt;
523     struct vif *vifp;
524     int *vifi;
525     u_char *ipoptions;
526     u_long tunnel_src;
527
528     if (ip->ip_hl < (IP_HDR_LEN + TUNNEL_LEN) >> 2 ||
529         (ipoptions = (u_char *) (ip + 1))[1] != IPOPT_LSRR) {
530         /* Packet arrived via a physical interface. */
531         tunnel_src = 0;
532     } else {
533         /*
534          * Packet arrived through a tunnel.
535          * A tunneled packet has a single NOP option and a
536          * two-element loose-source-and-record-route (LSRR)
537          * option immediately following the fixed-size part of
538          * the IP header. At this point in processing, the IP
539          * header should contain the following IP addresses:
540          *
541          * original source      - in the source address field
542          * destination group   - in the destination address field
543          * remote tunnel end-point - in the first element of LSRR
544          * one of this host's addrs - in the second element of LSRR
545          *
546          * NOTE: RFC-1075 would have the original source and
547          * remote tunnel end-point addresses swapped. However,
548          * that could cause delivery of ICMP error messages to
549          * innocent applications on intermediate routing
550          * hosts! Therefore, we hereby change the spec.
551          */
552         /* Verify that the tunnel options are well-formed. */
553         if (ipoptions[0] != IPOPT_NOP ||
554             ipoptions[2] != 11 || /* LSRR option length */
555             ipoptions[3] != 12 || /* LSRR address pointer */
556             (tunnel_src = *(u_long *) (&ipoptions[4])) == 0) {
557             mrtstat.mrtts_bad_tunnel++;
558             return (1);
559         }
560         /* Delete the tunnel options from the packet. */
561         ovbcopy((caddr_t) (ipoptions + TUNNEL_LEN), (caddr_t) ipoptions,
562                 (unsigned) (m->m_len - (IP_HDR_LEN + TUNNEL_LEN)));
563         m->m_len -= TUNNEL_LEN;
564         ip->ip_len -= TUNNEL_LEN;
565         ip->ip_hl -= TUNNEL_LEN >> 2;
566     }

```

**ip\_mforward** strips the tunnel options from the packet, consults the multicast routing table, and, in this example, forwards the packet on another tunnel and on the same *physical* interface on which it arrived, as shown by the thin arrows. This is OK

because the multicast routing tables are based on the *virtual* interfaces, not the physical interfaces.

In [Figure 14.36](#) we assume that the physical interface is a member of the destination group, so `ip_output` passes the datagram to `ip_mloopback`, which queues it for processing by `ipintr` (the thick arrows). The packet is passed to `ip_mforward` again, where it is discarded ([Exercise 14.4](#)). `ip_mforward` returns 0 this time (because the packet arrived on a physical interface), so `ipintr` considers and accepts the datagram for input processing.

We show the multicast forwarding code in three parts:

- tunnel input processing ([Figure 14.37](#)),
- forwarding eligibility ([Figure 14.39](#)), and
- forward to outgoing interfaces ([Figure 14.40](#)).

The two arguments to ip\_mforward are a pointer to the mbuf chain containing the datagram; and a pointer to the ifnet structure of the receiving interface.

## Arrival on physical interface

527-530

To distinguish between a multicast datagram arriving on a physical interface and a tunneled datagram arriving on the same physical interface, the IP header is examined for the characteristic LSRR option. If the header is too small to contain the option, or if the options don't start with a NOP followed by an LSRR option, it is assumed that the datagram arrived on a physical interface and tunnel\_src is set to 0.

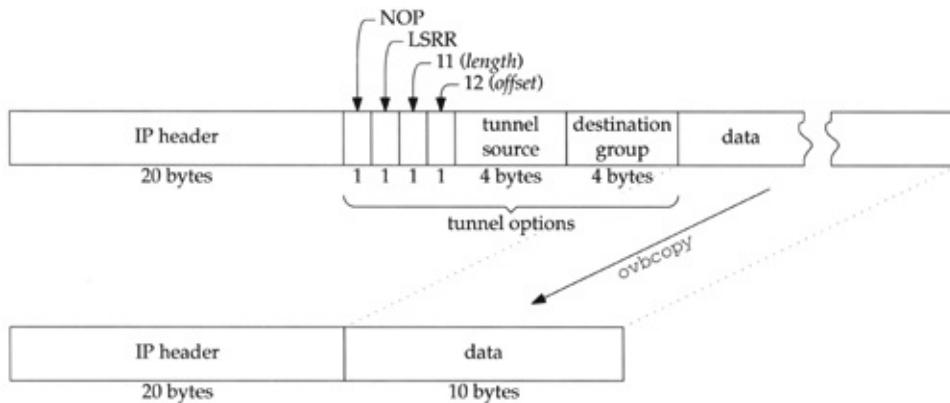
## Arrival on a tunnel

531-558

If the datagram looks as though it arrived on a tunnel, the options are verified to

make sure they are well formed. If the options are not well formed for a multicast tunnel, `ip_mforward` returns 1 to indicate that the datagram should be discarded. [Figure 14.38](#) shows the organization of the tunnel options.

**Figure 14.38. Multicast tunnel options.**



In [Figure 14.38](#) we assume there are no other options in the datagram, although that is not required. Any other IP options will appear after the LSRR option, which is always inserted before any other options by the multicast router at the start of the tunnel.

## Delete tunnel options

## 559-565

If the options are OK, they are removed from the datagram by shifting the remaining options and data forward and adjusting `m_len` in the mbuf header and `ip_len` and `ip_hl` in the IP header ([Figure 14.38](#)).

`ip_mforward` often uses `tunnel_source` as its return value, which is only nonzero when the datagram arrives on a tunnel. When `ip_mforward` returns a nonzero value, the caller discards the datagram. For `ipintr` this means that a datagram that arrives on a tunnel is passed to `ip_mforward` and discarded by `ipintr`. The forwarding code strips out the tunnel information, duplicates the datagram, and sends the datagrams with `ip_output`, which calls `ip_mloopback` if the interface is a member of the destination group.

The next part of `ip_mforward`, shown in [Figure 14.39](#), discards the datagram if it is ineligible for forwarding.

## Figure 14.39. ip\_mforward function: forwarding eligibility checks.

```
ip_mroute.c
566  /*
567   * Don't forward a packet with time-to-live of zero or one,
568   * or a packet destined to a local-only group.
569   */
570  if (ip->ip_ttl <= 1 ||
571      ntohl(ip->ip_dst.s_addr) <= INADDR_MAX_LOCAL_GROUP)
572      return ((int) tunnel_src);

573  /*
574   * Don't forward if we don't have a route for the packet's origin.
575   */
576  if (!(rt = mrtfind(ip->ip_src))) {
577      mrtstat.mrts_no_route++;
578      return ((int) tunnel_src);
579  }
580  /*
581   * Don't forward if it didn't arrive from the parent vif for its origin.
582   */
583  vifi = rt->mrt_parent;
584  if (tunnel_src == 0) {
585      if ((viftable[vifi].v_flags & VIFF_TUNNEL) ||
586          viftable[vifi].v_ifp != ifp)
587          return ((int) tunnel_src);
588  } else {
589      if (!(viftable[vifi].v_flags & VIFF_TUNNEL) ||
590          viftable[vifi].v_rmt_addr.s_addr != tunnel_src)
591          return ((int) tunnel_src);
592  }
```

---

```
ip_mroute.c
```

## Expired TTL or local multicast

566-572

If ip\_ttl is 0 or 1, the datagram has reached the end of its lifetime and is not forwarded. If the destination group is less than or equal to INADDR\_MAX\_LOCAL\_GROUP (the 224.0.0.x groups, [Figure 12.1](#)), the datagram is not allowed beyond the local

network and is not forwarded. In either case, tunnel\_src is returned to the caller.

Version 3.3 of mrouted supports administrative scoping of certain destination groups. An interface can be configured to discard datagrams addressed to these groups, similar to the automatic scoping of the 224.0.0.x groups.

## No route available

573-579

If mrtfind cannot locate a route based on the *source* address of the datagram, the function returns. Without a route, the multicast router cannot determine to which interfaces the datagram should be forwarded. This might occur, for example, when the multicast datagrams arrive before the multicast routing table has been updated by mrouted.

## Arrived on unexpected interface

580-592

If the datagram arrived on a physical interface but was expected to arrive on a tunnel or on a different physical interface, `ip_mforward` returns. If the datagram arrived on a tunnel but was expected to arrive on a physical interface or on a different tunnel, `ip_mforward` returns. A datagram may arrive on an unexpected interface when the routing tables are in transition because of changes in the group membership or in the physical topology of the network.

The final part of `ip_mforward` ([Figure 14.40](#)) sends the datagram on each of the outgoing interfaces specified in the multicast route entry.

**Figure 14.40. `ip_mforward` function:  
forwarding.**

---

```

593  /*
594   * For each vif, decide if a copy of the packet should be forwarded.
595   * Forward if:
596   *   - the ttl exceeds the vif's threshold AND
597   *   - the vif is a child in the origin's route AND
598   *   - ( the vif is not a leaf in the origin's route OR
599   *       the destination group has members on the vif )
600   *
601   * (This might be speeded up with some sort of cache -- someday.)
602   */
603 for (vifp = viftable, vifi = 0; vifi < numvifs; vifp++, vifi++) {
604     if (ip->ip_ttl > vifp->v_threshold &&
605         VIFM_ISSET(vifi, rt->mrt_children) &&
606         (!VIFM_ISSET(vifi, rt->mrt_leaves) ||
607          grplist_member(vifp, ip->ip_dst))) {
608       if (vifp->v_flags & VIFF_TUNNEL)
609         tunnel_send(m, vifp);
610       else
611         phyint_send(m, vifp);
612     }
613   }
614   return ((int) tunnel_src);
615 }
```

---

ip\_mroute.c

## 593-615

For each interface in viftable, a datagram is sent on the interface if

- the datagram's TTL is greater than the multicast threshold for the interface,
- the interface is a child interface for the route, and
- the interface is not connected to a leaf network.

If the interface is a leaf, the datagram is output only if there is a member of the destination group on the network (i.e., grplist\_member returns a nonzero value).

tunnel\_send forwards the datagram on tunnel interfaces; phyint\_send is used for physical interfaces.

## phyint\_send Function

To send a multicast datagram on a physical interface, phyint\_send (Figure 14.41) specifies the output interface explicitly in the ip\_moptions structure it passes to ip\_output.

**Figure 14.41. phyint\_send function.**

```
ip_mroute.c
616 static void
617 phyint_send(m, vifp)
618 struct mbuf *m;
619 struct vif *vifp;
620 {
621     struct ip *ip = mtod(m, struct ip *);
622     struct mbuf *mb_copy;
623     struct ip_moptions *imo;
624     int     error;
625     struct ip_moptions simo;

626     mb_copy = m_copy(m, 0, M_COPYALL);
627     if (mb_copy == NULL)
628         return;

629     imo = &simo;
630     imo->imo_multicast_ifp = vifp->v_ifp;
631     imo->imo_multicast_ttl = ip->ip_ttl - 1;
632     imo->imo_multicast_loop = 1;

633     error = ip_output(mb_copy, NULL, NULL, IP_FORWARDING, imo);
634 }
```

616-634

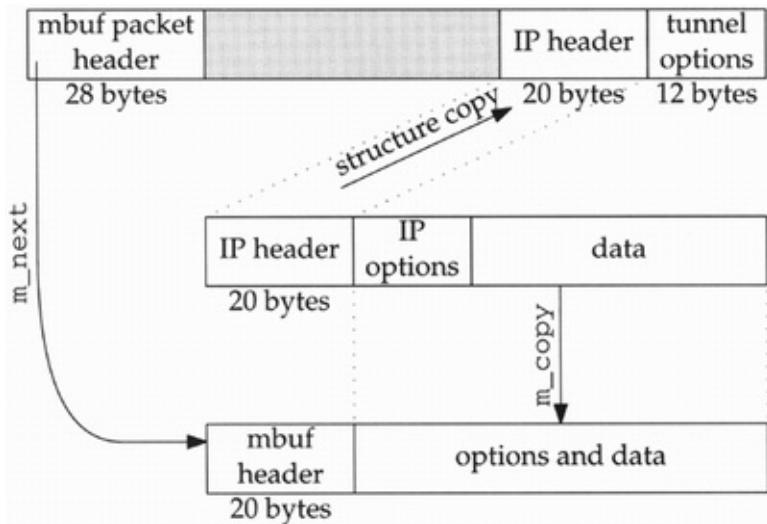
`m_copy` duplicates the outgoing datagram. The `ip_moptions` structure is set to force the datagram to be transmitted on the selected interface. The TTL value is decremented, and multicast loopback is enabled.

The datagram is passed to `ip_output`. The `IP_FORWARDING` flag avoids an infinite loop, where `ip_output` calls `ip_mforward` again.

## tunnel\_send Function

To send a datagram on a tunnel, `tunnel_send` ([Figure 14.43](#)) must construct the appropriate tunnel options and insert them in the header of the outgoing datagram. [Figure 14.42](#) shows how `tunnel_send` prepares a packet for the tunnel.

**Figure 14.42. Inserting tunnel options.**



**Figure 14.43. `tunnel_send` function:  
verify and allocate new header.**

---

```

635 static void
636 tunnel_send(m, vifp)
637 struct mbuf *m;
638 struct vif *vifp;
639 {
640     struct ip *ip = mtod(m, struct ip *);
641     struct mbuf *mb_copy, *mb_opts;
642     struct ip *ip_copy;
643     int error;
644     u_char *cp;

645     /*
646      * Make sure that adding the tunnel options won't exceed the
647      * maximum allowed number of option bytes.
648      */
649     if (ip->ip_hl > (60 - TUNNEL_LEN) >> 2) {
650         mrtstat.mrts_cant_tunnel++;
651         return;
652     }
653     /*
654      * Get a private copy of the IP header so that changes to some
655      * of the IP fields don't damage the original header, which is
656      * examined later in ip_input.c.
657      */
658     mb_copy = m_copy(m, IP_HDR_LEN, M_COPYALL);
659     if (mb_copy == NULL)
660         return;
661     MGETHDR(mb_opts, M_DONTWAIT, MT_HEADER);
662     if (mb_opts == NULL) {
663         m_free(mb_copy);
664         return;
665     }
666     /*
667      * Make mb_opts be the new head of the packet chain.
668      * Any options of the packet were left in the old packet chain head
669      */
670     mb_opts->m_next = mb_copy;
671     mb_opts->m_len = IP_HDR_LEN + TUNNEL_LEN;
672     mb_opts->m_data += MSIZE - mb_opts->m_len;

```

---

ip\_mroute.c

## Will the tunnel options fit?

635-652

If there is no room in the IP header for the tunnel options, tunnel\_send returns immediately and the datagram is not forwarded on the tunnel. It may be forwarded on other interfaces.

## Duplicate the datagram and allocate mbuf for new header and tunnel options

653-672

In the call to `m_copy`, the starting offset for the copy is 20 (`IP_HDR_LEN`). The resulting mbuf chain contains the options and data for the datagram but not the IP header. `mb_opts` points to a new datagram header allocated by `MGETHDR`. The datagram header is prepended to `mb_copy`. Then `m_len` and `m_data` are adjusted to accommodate an IP header and the tunnel options.

The second half of `tunnel_send`, shown in [Figure 14.44](#), modifies the headers of the outgoing packet and sends the packet.

**Figure 14.44. `tunnel_send` function:  
construct headers and send.**

```
673     ip_copy = mtod(mb_opts, struct ip *);
674     /*
675      * Copy the base ip header to the new head mbuf.
676      */
677     *ip_copy = *ip;
678     ip_copy->ip_ttl--;
679     ip_copy->ip_dst = vifp->v_rmt_addr;      /* remote tunnel end-point */
680     /*
681      * Adjust the ip header length to account for the tunnel options.
682      */
683     ip_copy->ip_hl += TUNNEL_LEN >> 2;
684     ip_copy->ip_len += TUNNEL_LEN;
685     /*
686      * Add the NOP and LSRR after the base ip header
687      */
688     cp = (u_char *) (ip_copy + 1);
689     *cp++ = IPOPT_NOP;
690     *cp++ = IPOPT_LSRR;
691     *cp++ = 11;                      /* LSRR option length */
692     *cp++ = 8;                      /* LSSR pointer to second element */
693     *(u_long *) cp = vifp->v_lcl_addr.s_addr; /* local tunnel end-point */
694     cp += 4;
695     *(u_long *) cp = ip->ip_dst.s_addr;    /* destination group */
696     error = ip_output(mb_opts, NULL, NULL, IP_FORWARDING, NULL);
697 }
```

ip\_mroute.c

## Modify IP header

673-679

The original IP header is copied from the original mbuf chain into the newly allocated mbuf header. The TTL in the header is decremented, and the destination is changed to be the other end of the tunnel.

## Construct tunnel options

680-664

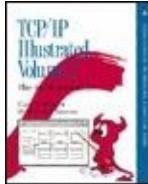
ip\_h1 and ip\_len are adjusted to accommodate the tunnel options. The tunnel options are placed just after the IP header: a NOP, followed by the LSRR code, the length of the LSRR option (11 bytes), and a pointer to the *second* address in the option (8 bytes). The source route consists of the local tunnel end point followed by the destination group ([Figure 14.13](#)).

## Send the tunneled datagram

665-697

ip\_output sends the datagram, which now looks like a unicast datagram with an LSRR option since the destination address is the unicast address of the other end of the tunnel. When it reaches the other end of the tunnel, the tunnel options are stripped off and the datagram is forwarded at that point, possibly through additional tunnels.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.9 Cleanup: ip\_mrouter\_done Function

When mrouted shuts down, it issues the DVMRP\_DONE command, which is handled by the ip\_mrouter\_done function shown in Figure 14.45.

**Figure 14.45. ip\_mrouter\_done function:  
DVMRP\_DONE command.**

---

```

161 int
162 ip_mrouter_done()
163 {
164     vifi_t  vifi;
165     int     i;
166     struct ifnet *ifp;
167     int     s;
168     struct ifreq ifr;
169     s = splnet();
170     /*
171      * For each phyint in use, free its local group list and
172      * disable promiscuous reception of all IP multicasts.
173      */
174     for (vifi = 0; vifi < numvifs; vifi++) {
175         if (vitable[vifi].v_lcl_addr.s_addr != 0 &&
176             !(vitable[vifi].v_flags & VIFF_TUNNEL)) {
177             if (vitable[vifi].v_lcl_grps)
178                 free(vitable[vifi].v_lcl_grps, M_MRTABLE);
179             satosin(&ifr.ifr_addr)->sin_family = AF_INET;
180             satosin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
181             ifp = vitable[vifi].v_ifp;
182             (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
183         }
184     }
185     bzero((caddr_t) vitable, sizeof(vitable));
186     numvifs = 0;
187     /*
188      * Free any multicast route entries.
189      */
190     for (i = 0; i < MRTHASHSIZ; i++)
191         if (mrtable[i])
192             free(mrtable[i], M_MRTABLE);
193     bzero((caddr_t) mrtable, sizeof(mrtable));
194     cached_mrt = NULL;
195     ip_mrouter = NULL;
196     splx(s);
197     return (0);
198 }
```

---

ip\_mroute.c

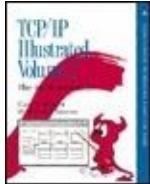
## 161-186

This function runs at splnet to avoid any interaction with the multicast forwarding code. For every physical multicast interface, the list of local groups is released and the SIOCDELMULTI command is issued to stop receiving multicast datagrams ([Exercise 14.3](#)). The entire vitable array is cleared by bzero and numvifs is set to 0.

## 187-198

Every active entry in the multicast routing table is released, the entire table is cleared with bzero, the cache is cleared, and ip\_mrouter is reset.

Each entry in the multicast routing table may be the first in a linked list of entries. This code introduces a memory leak by releasing only the first entry in the list.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 14. IP Multicast Routing

### 14.10 Summary

In this chapter we described the general concept of internetwork multicasting and the specific functions within the Net/3 kernel that support it. We did not discuss the implementation of mrouted, but the source is readily available for the interested reader.

We described the virtual interface table and the differences between a physical interface and a tunnel, as well as the LSRR options used to implement tunnels in Net/3.

We illustrated the RPB, TRPB, and RPM algorithms and described the kernel tables

used to forward multicast datagrams according to TRPB. The concept of parent and leaf networks was also discussed.

## Exercises

**14.1** In [Figure 14.25](#), how many multicast routes are needed?

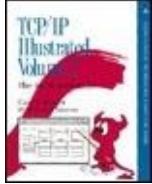
Why is the update to the group membership cache in [Figure 14.23](#) protected by splnet and splx?

What happens when SIOCDELMULTI is issued for an interface that has explicitly joined a multicast group with the IP\_ADD\_MEMBERSHIP option?

When a datagram arrives on a tunnel and is accepted by ip\_mforward, it may be looped back by ip\_output when it is forwarded to a physical interface. Why does ip\_mforward

discard the looped-back packet when it arrives on the physical interface?

**14.5** Redesign the group address cache to increase its effectiveness.



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 15. Socket Layer

Section 15.1. Introduction

Section 15.2. Code Introduction

Section 15.3. socket Structure

Section 15.4. System Calls

Section 15.5. Processes, Descriptors,  
and Sockets

Section 15.6. socket System Call

Section 15.7. getsock and sockargs  
Functions

Section 15.8. bind System Call

Section 15.9. listen System Call

Section 15.10. tsleep and wakeup  
Functions

Section 15.11. accept System Call

Section 15.12. sonewconn and

## soisconnected Functions

Section 15.13. connect System call

Section 15.14. shutdown System Call

Section 15.15. close System Call

Section 15.16. Summary

---

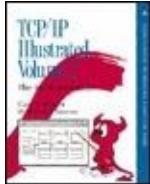
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.1 Introduction

This chapter is the first of three that cover the socket-layer code in Net/3. The socket abstraction was introduced with the 4.2BSD release in 1983 to provide a uniform interface to network and interprocess communication protocols. The Net/3 release discussed here is based on the 4.3BSD Reno version of sockets, which is slightly different from the earlier 4.2 releases used by many Unix vendors.

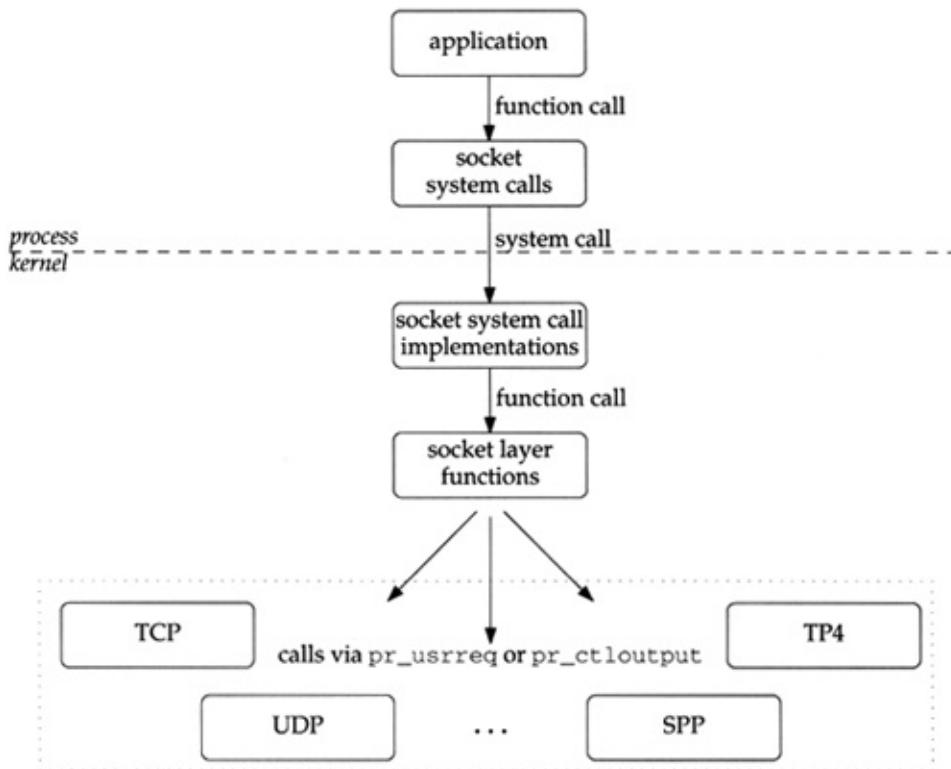
As described in [Section 1.7](#), the socket layer maps protocol-independent requests from a process to the protocol-specific implementation selected when the socket was created.

To allow standard Unix I/O system calls such as read and write to operate with network connections, the filesystem and networking facilities in BSD releases are integrated at the system call level. Network connections represented by sockets are accessed through a descriptor (a small integer) in the same way an open file is accessed through a descriptor. This allows the standard filesystem calls such as read and write, as well as network-specific system calls such as sendmsg and recvmsg, to work with a descriptor associated with a socket.

Our focus is on the implementation of sockets and the associated system calls and not on how a typical program might use the socket layer to implement network applications. For a detailed discussion of the process-level socket interface and how to program network applications see [\[Stevens 1990\]](#) and [\[Rago 1993\]](#).

[Figure 15.1](#) shows the layering between the socket interface in a process and the protocol implementation in the kernel.

**Figure 15.1. The socket layer converts generic requests to specific protocol operations.**



## splnet Processing

The socket layer contains many paired calls to splnet and splx. As discussed in [Section 1.12](#), these calls protect code that accesses data structures shared between the socket layer and the protocol-processing layer. Without calls to splnet, a software interrupt that initiates protocol

processing and changes the shared data structures will confuse the socket-layer code when it resumes.

We assume that readers understand these calls and we rarely point them out in our discussion.

---

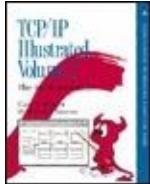
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

## 15.2 Code Introduction

The three files listed in [Figure 15.2](#) are described in this chapter.

### Figure 15.2. Files discussed in this chapter.

File	Description
sys/socketvar.h	socket structure definitions
kern/uipc_syscalls.c	system call implementation
kern/uipc_socket.c	socket-layer functions

## Global Variables

The two global variable covered in this chapter are described in [Figure 15.3](#).

## Figure 15.3. Global variable introduced in this chapter.

Variable	Datatype	Description
socketops sysent	struct fileops struct sysent[]	socket implementation of I/O system calls array of system call entries

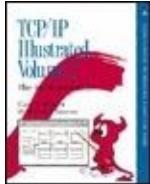
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.3 socket Structure

A socket represents one end of a communication link and holds or points to all the information associated with the link. This information includes the protocol to use, state information for the protocol (which includes source and destination addresses), queues of arriving connections, data buffers, and option flags. [Figure 15.5](#) shows the definition of a socket and its associated buffers.

41-42

`so_type` is specified by the process creating a socket and identifies the communication semantics to be supported

by the socket and the associated protocol. `so_type` shares the same values as `pr_type` shown in [Figure 7.8](#). For UDP, `so_type` would be `SOCK_DGRAM` and for TCP it would be `SOCK_STREAM`.

43

`so_options` is a collection of flags that modify the behavior of a socket. [Figure 15.4](#) describes the flags.

**Figure 15.4. `so_options` values.**

<code>so_options</code>	Kernel only	Description
<code>SO_ACCEPTCONN</code>	*	socket accepts incoming connections
<code>SO_BROADCAST</code>		socket can send broadcast messages
<code>SO_DEBUG</code>		socket records debugging information
<code>SO_DONTROUTE</code>		output operations bypass routing tables
<code>SO_KEEPALIVE</code>		socket probes idle connections
<code>SO_OOBINLINE</code>		socket keeps out-of-band data inline
<code>SO_REUSEADDR</code>		socket can reuse a local address
<code>SO_REUSEPORT</code>		socket can reuse a local address and port
<code>SO_USELOOPBACK</code>		routing domain sockets only; sending process receives its own routing requests

A process can modify all the socket options with the `getsockopt` and `setsockopt` system calls except `SO_ACCEPTCONN`, which is set by the kernel when the `listen` system call is issued on the socket.

## Figure 15.5. struct socket definition.

```
socketvar.h
41 struct socket {
42     short    so_type;          /* generic type, Figure 7.8 */
43     short    so_options;       /* from socket call, Figure 15.4 */
44     short    so_linger;        /* time to linger while closing */
45     short    so_state;         /* internal state flags, Figure 15.6 */
46     caddr_t  so_pcb;          /* protocol control block */
47     struct protosw *so_proto; /* protocol handle */
48 /*
49 * Variables for connection queueing.
50 * Socket where accepts occur is so_head in all subsidiary sockets.
51 * If so_head is 0, socket is not related to an accept.
52 * For head socket so_q0 queues partially completed connections,
53 * while so_q is a queue of connections ready to be accepted.
54 * If a connection is aborted and it has so_head set, then
55 * it has to be pulled out of either so_q0 or so_q.
56 * We allow connections to queue up based on current queue lengths
57 * and limit on number of queued connections for this socket.
58 */
59     struct socket *so_head;   /* back pointer to accept socket */
60     struct socket *so_q0;    /* queue of partial connections */
61     struct socket *so_q;    /* queue of incoming connections */
62     short    so_qolen;      /* partials on so_q0 */
63     short    so_qlen;        /* number of connections on so_q */
64     short    so_qlimit;     /* max number queued connections */
65     short    so_timeo;      /* connection timeout */
66     u_short  so_error;     /* error affecting connection */
67     pid_t    so_pgid;       /* pgid for signals */
68     u_long   so_oobmark;    /* chars to oob mark */
69 /*
70 * Variables for socket buffering.
71 */
72     struct sockbuf {
73         u_long   sb_cc;        /* actual chars in buffer */
74         u_long   sb_hiwat;     /* max actual char count */
75         u_long   sb_mbent;     /* chars of mbufs used */
76         u_long   sb_mbmax;     /* max chars of mbufs to use */
77         long    sb_lowat;      /* low water mark */
78         struct mbuf *sb_mb;    /* the mbuf chain */
79         struct selinfo sb_sel; /* process selecting read/write */
80         short   sb_flags;      /* Figure 16.5 */
81         short   sb_timeo;     /* timeout for read/write */
82     } so_rcv, so_snd;
83     caddr_t  so_tpcb;        /* Wisc. protocol control block XXX */
84     void (*so_upcall) (struct socket * so, caddr_t arg, int waitf);
85     caddr_t  so_upcallarg;   /* Arg for above */
86 };
```

socketvar.h

44

so\_linger is the time in clock ticks that a socket waits for data to drain while closing a connection ([Section 15.15](#)).

`so_state` represents the internal state and additional characteristics of the socket. [Figure 15.6](#) lists the possible values for `so_state`.

**Figure 15.6. `so_state` values.**

<code>so_state</code>	Kernel only	Description
<code>SS_ASYNC</code> <code>SS_NBIO</code>		socket should send asynchronous notification of I/O events socket operations should not block the process
<code>SS_CANTRCVMORE</code> <code>SS_CANTSENDMORE</code> <code>SS_ISCONFIRMING</code> <code>SS_ISCONNECTED</code> <code>SS_ISCONNECTING</code> <code>SS_ISDISCONNECTING</code> <code>SS_NOFDREF</code> <code>SS_PRIV</code> <code>SS_RCVATMARK</code>	• • • • • • • • • •	socket cannot receive more data from peer socket cannot send more data to peer socket is negotiating a connection request socket is connected to a foreign socket socket is connecting to a foreign socket socket is disconnecting from peer socket is not associated with a descriptor socket was created by a process with superuser privileges process has consumed all data received before the most recent out-of-band data was received

In [Figure 15.6](#), the middle column shows that `SS_ASYNC` and `SS_NBIO` can be changed explicitly by a process by the `fcntl` and `ioctl` system calls. The other flags are implicitly changed by the process during the execution of system calls. For example, if the process calls `connect`, the `SS_ISCONNECTED` flag is set by the kernel when the connection is established.

## **SS\_NBIO and SS\_ASYNC Flags**

By default, a process blocks waiting for resources when it makes an I/O request. For example, a read system call on a socket blocks if there is no data available from the network. When the data arrives, the process is unblocked and read returns. Similarly, when a process calls write, the kernel blocks the process until space is available in the kernel for the data. If SS\_NBIO is set, the kernel does not block a process during I/O on the socket but instead returns the error code EWOULDBLOCK.

If SS\_ASYNC is set, the kernel sends the SIGIO signal to the process or process group specified by so\_pgid when the status of the socket changes for one of the following reasons:

- a connection request has completed,
- a disconnect request has been initiated,
- a disconnect request has completed,

- half of a connection has been shut down,
- data has arrived on a socket,
- data has been sent from a socket (i.e., the output buffer has free space), or
- an asynchronous error has occurred on a UDP or TCP socket.

46

`so_pcb` points to a protocol control block that contains protocol-specific state information and parameters for the socket. Each protocol defines its own control block structure, so `so_pcb` is defined to be a generic pointer. [Figure 15.7](#) lists the control block structures that we discuss.

### Figure 15.7. Protocol control blocks.

Protocol	Control block	Reference
UDP	<code>struct inpcb</code>	Section 22.3
TCP	<code>struct inpcb</code>	Section 22.3
	<code>struct tcpcb</code>	Section 24.5
ICMP, IGMP, raw IP	<code>struct inpcb</code>	Section 22.3
Route	<code>struct rawcb</code>	Section 20.3

so\_pcb never points to a tcpcb structure directly; see [Figure 22.1](#).

47

so\_proto points to the protosw structure of the protocol selected by the process during the socket system call ([Section 7.4](#)).

48-64

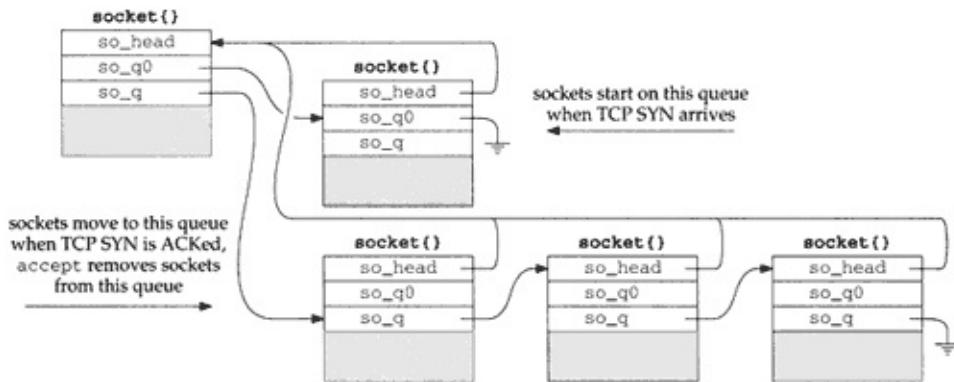
Sockets with SO\_ACCEPTCONN set maintain two connection queues. Connections that are not yet established (e.g., the TCP three-way handshake is not yet complete) are placed on the queue so\_q0. Connections that are established and are ready to be accepted (e.g., the TCP three-way handshake is complete) are placed on the queue so\_q. The lengths of the queues are kept in so\_q0len and so\_qlen. Each queued connection is represented by its own socket. so\_head in each queued socket points to the original socket with SO\_ACCEPTCONN set.

The maximum number of queued connections for a particular socket is

controlled by `so_qlimit`, which is specified by a process when it calls `listen`. The kernel silently enforces an upper limit of 5 (`SOMAXCONN`, [Figure 15.24](#)) and a lower limit of 0. A somewhat obscure formula shown with [Figure 15.29](#) uses `so_qlimit` to control the number of queued connections.

[Figure 15.8](#) illustrates a queue configuration in which three connections are ready to be accepted and one connection is being established.

## Figure 15.8. Socket connection queues.



65

`so_timeo` is a *wait channel* ([Section 15.10](#)) used during `accept`, `connect`, and `close` processing.

66

so\_error holds an error code until it can be reported to a process during the next system call that references the socket.

67

If SS\_ASYNC is set for a socket, the SIGIO signal is sent to the process (if so\_pgid is greater than 0) or to the progress group (if so\_pgid is less than 0). so\_pgid can be changed or examined with the SIOCSPGRP and SIOCGPGRP ioctl commands. For more information about process groups see [[Stevens 1992](#)].

68

so\_oobmark identifies the point in the input data stream at which out-of-band data was most recently received. [Section 16.11](#) discusses socket support for out-of-band data and [Section 29.7](#) discusses the semantics of out-of-band data in TCP.

69-82

Each socket contains two data buffers,

`so_rcv` and `so_snd`, used to buffer incoming and outgoing data. These are structures contained within the socket structure, not pointers to structures. We describe the organization and use of the socket buffers in [Chapter 16](#).

83-86

`so_tpcb` is not used by Net/3. `so_upcall` and `so_upcallarg` are used only by the NFS software in Net/3.

NFS is unusual. In many ways it is a process-level application that has been moved into the kernel. The `so_upcall` mechanism triggers NFS input processing when data is added to a socket receive buffer. The `tsleep` and `wakeup` mechanism is inappropriate in this case, since the NFS protocol executes within the kernel, not as a process.

The files `socketvar.h` and `uipc_socket2.c` define several macros and functions that simplify the socket-layer code. [Figure 15.9](#) summarizes them.

## Figure 15.9. Socket macros and functions.

Name	Description
<code>sosendallatonce</code>	Does the protocol associated with <code>so</code> require each send system call to result in a single protocol request?  <code>int sosendallatonce(struct socket *so);</code>
<code>soisconnecting</code>	Set the socket state to <code>SS_ISCONNECTING</code> .  <code>int soisconnecting(struct socket *so);</code>
<code>soisconnected</code>	See Figure 15.30.
<code>soreadable</code>	Will a read on <code>so</code> return information without blocking?  <code>int soreadable(struct socket *so);</code>
<code>sowriteable</code>	Will a write on <code>so</code> return without blocking?  <code>int sowriteable(struct socket *so);</code>
<code>socantsendmore</code>	Set the <code>SS_CANTSENDMORE</code> flag. Wake up any processes sleeping on the send buffer.  <code>int socantsendmore(struct socket *so);</code>
<code>socantrcvmore</code>	Set the <code>SS_CANTRCVMORE</code> flag. Wake up processes sleeping on the receive buffer.  <code>int socantrcvmore(struct socket *so);</code>
<code>sodisconnect</code>	Issue the <code>PRU_DISCONNECT</code> request.  <code>int sodisconnect(struct socket *so);</code>
<code>soisdisconnecting</code>	Clear the <code>SS_ISCONNECTING</code> flag. Set <code>SS_ISDISCONNECTING</code> , <code>SS_CANTRCVMORE</code> , and <code>SS_CANTSENDMORE</code> flags. Wake up any processes selecting on the socket.  <code>int soisdisconnecting(struct socket *so);</code>
<code>soisdisconnected</code>	Clear the <code>SS_ISCONNECTING</code> , <code>SS_ISCONNECTED</code> , and <code>SS_ISDISCONNECTING</code> flags. Set the <code>SS_CANTRCVMORE</code> and <code>SS_CANTSENDMORE</code> flags. Wake up any processes selecting on the socket or waiting for <code>close</code> to complete.  <code>int soisdisconnected(struct socket *so);</code>
<code>soqinsque</code>	Insert <code>so</code> on a queue associated with <code>head</code> . If <code>q</code> is 0, the socket is added to the end of <code>so_q0</code> , which holds incomplete connections. Otherwise, the socket is added to the end of <code>so_q</code> , which holds connections that are ready to be accepted. Net/1 incorrectly placed sockets at the front of the queue.  <code>int soqinsque(struct socket *head, struct socket *so, int q);</code>
<code>soqremque</code>	Remove <code>so</code> from the queue identified by <code>q</code> . The socket queues are located by following <code>so-&gt;so_head</code> .  <code>int soqremque(struct socket *so, int q);</code>

## Chapter 15. Socket Layer

---

### 15.4 System Calls

A process interacts with the kernel through a collection of *system calls*. Before showing the system calls themselves, we will look at the system call mechanism itself.

The transfer of execution from a process to the kernel is controlled by the system call mechanism. In this section, we will look at the implementation of Net/3 to illustrate implementation details.

In BSD kernels, each system call is numbered and mapped to control to a single kernel function when the process makes the call. In particular, the number of the particular system call is identified as an integer argument to the system call. In the Net/3 implementation, sysent is that function. Using the sysent table to locate the sysent structure for the requested system call, the kernel creates a sysent structure:

```
struct sysent {  
    int sy_nargs; /* number of arguments */  
    ...  
};
```

```
        int (*sy_call) () ;           /* in
} ;                                /* sy
```

Here are several entries from the sysent array,

```
struct sysent sysent[] = {
    /* ... */
    { 3, recvmsg },
    { 3, sendmsg },
    { 6, recvfrom },
    { 3, accept },
    { 3, getpeername },
    { 3, getsockname },
    /* ... */
}
```

For example, the recvmsg system call is the 27 three arguments, and is implemented by the re

syscall copies the arguments from the calling p array to hold the results of the system call, whi system call completes, syscall dispatches contr the system call. In the 386 implementation, thi

```
struct sysent *callp;
error = (*callp->sy_call) (p, arc
```

where callp is a pointer to the relevant sysent s-table entry for the process that made the system call as an array of 32-bit words, and hold the return value of the system call. When function within the kernel called by syscall, not the application.

syscall expects the system call function (i.e., w errors occurred and a nonzero error code other the values in rval back to the process as the ret made by the application). If an error occurs, sy the error code to the process in a machine-dep available to the process in the external variable application returns 1 or a null pointer to indicat

The 386 implementation sets the carry bit to in an error code. The system call stub in the proce or a null pointer to the application. If the carry syscall is returned by the stub.

To summarize, a function implementing a syste syscall function, and a second (found in rval) th when no error occurs.

## Example

The prototype for the socket system call is:

```
int socket (int domain, int type
```

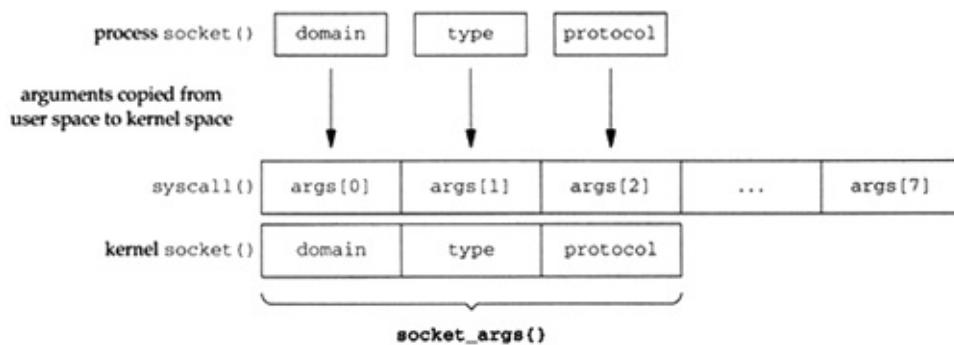
The prototype for the kernel function that implements

```
struct socket_args {  
    int domain;  
    int type;  
    int protocol;  
};
```

```
socket(struct proc *p, struct sc
```

When an application calls socket, the process passes arguments to the kernel with the system call mechanism, syscall. The application bit values and passes a pointer to the array as arguments copied from user space to kernel space of socket. The kernel version of socket treats the arguments as a socket\_args structure. [Figure 15.10](#) illustrates the mapping between the user-space arguments and the kernel-space socket\_args structure.

**Figure 15.10. socket arguments mapping**



As illustrated by socket, each kernel function takes its arguments not as a pointer to an array of 32-bit words, but as individual system call.

The implicit cast is legal only in traditional K&R C if no prototype is in effect. If a prototype is in effect, the compiler

syscall prepares the return value of 0 before executing the system call. If no error occurs, the system call function can return 0 to the process.

## System Call Summary

Figure 15.11 summarizes the system calls relevant to networking.

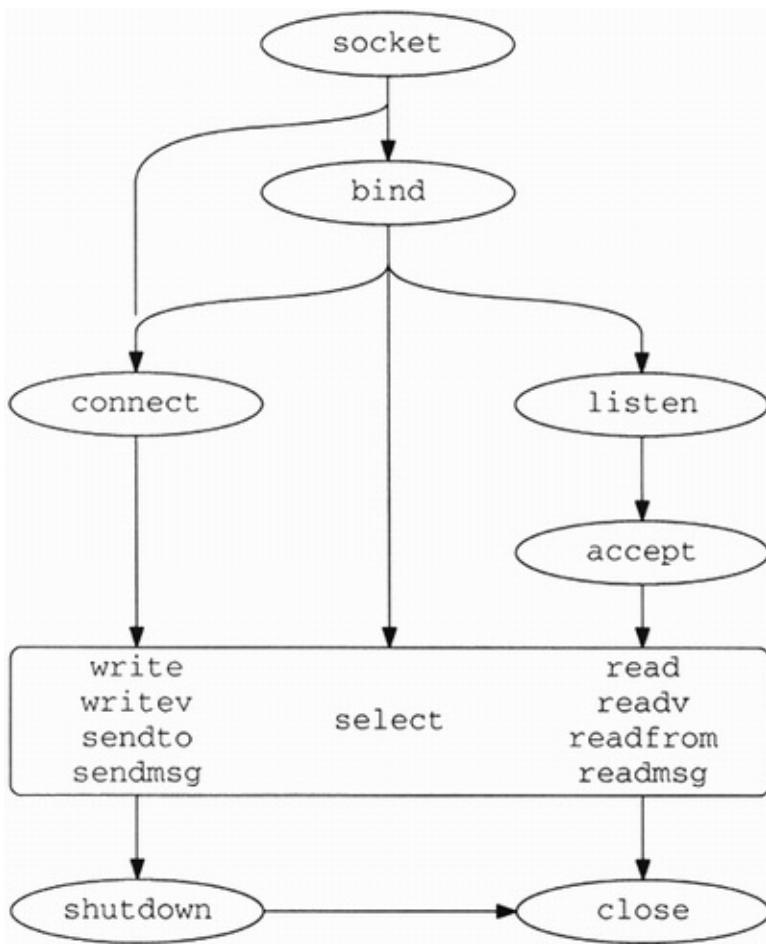
**Figure 15.11. Networking**

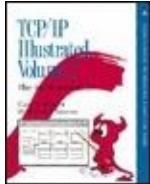
Category	Name	Function
setup	socket	create a new unnamed socket within a specified communication domain
	bind	assign a local address to a socket
server	listen accept	prepare a socket to accept incoming connections wait for and accept connections
client	connect	establish a connection to a foreign socket
input	read	receive data into a single buffer
	readv	receive data into multiple buffers
	recv	receive data specifying options
	recvfrom	receive data and address of sender
	recvmsg	receive data into multiple buffers, control information, and receive the address of sender; specify receive options
output	write	send data from a single buffer
	writev	send data from multiple buffers
	send	send data specifying options
	sendto	send data to specified address
	sendmsg	send data from multiple buffers and control information to a specified address; specify send options
I/O	select	wait for I/O conditions
termination	shutdown	terminate connection in one or both directions
	close	terminate connection and release socket
administration	fcntl	modify I/O semantics
	ioctl	miscellaneous socket operations
	setsockopt	set socket or protocol options
	getsockopt	get socket or protocol options
	getsockname	get local address assigned to socket
	getpeername	get foreign address assigned to socket

We present the setup, server, client, and termination system calls are discussed in [Chapter 16](#) and [Chapter 17](#).

[Figure 15.12](#) shows the sequence in which an application can call the system calls in the large box in an appropriate order. The diagram also shows some valid transitions are not included in the sequence.

**Figure 15.12. Network system call sequence**





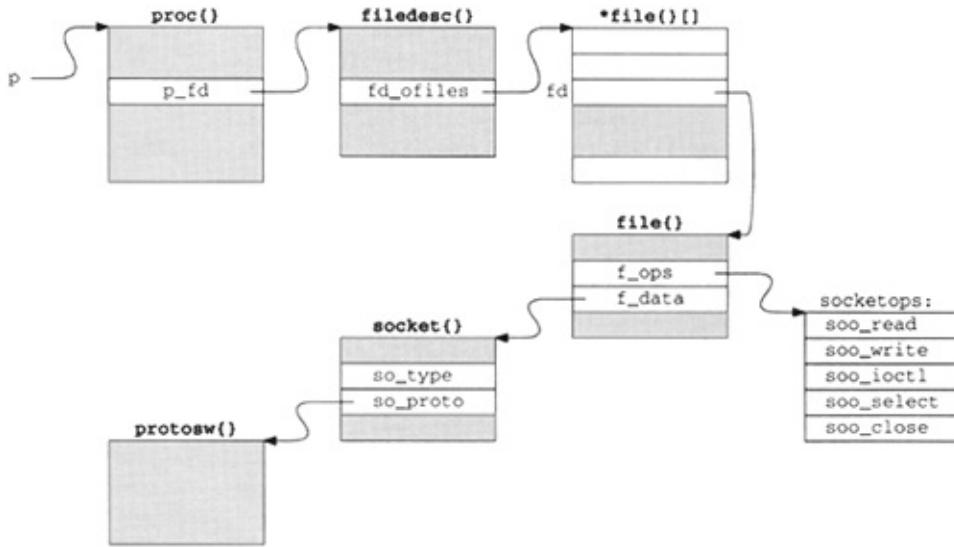
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.5 Processes, Descriptors, and Sockets

Before describing the socket system calls, we need to discuss the data structures that tie together processes, descriptors, and sockets. [Figure 15.13](#) shows the structures and members relevant to our discussion. A more complete explanation of the file structures can be found in [\[Leffler et al. 1989\]](#).

**Figure 15.13. Process, file, and socket structures.**



The first argument to a function implementing a system call is always `p`, a pointer to the `proc` structure of the calling process. The `proc` structure represents the kernel's notion of a process. Within the `proc` structure, `p_fd` points to a `filedesc` structure, which manages the descriptor table pointed to by `fd_ofiles`. The descriptor table is dynamically sized and consists of an array of pointers to `file` structures. Each `file` structure describes a single open file and can be shared between multiple processes.

Only a single `file` structure is shown in [Figure 15.13](#). It is accessed by `p->p_fd->fd_ofiles[fd]`. Within the `file` structure, two members are of interest to us: `f_ops`

and `f_data`. The implementation of I/O system calls such as read and write varies according to what type of I/O object is associated with a descriptor. `f_ops` points to a `fileops` structure containing a list of function pointers that implement the read, write, ioctl, select, and close system calls for the associated I/O object. [Figure 15.13](#) shows `f_ops` pointing to a global `fileops` structure, `socketops`, which contains pointers to the functions for sockets.

`f_data` points to private data used by the associated I/O object. For sockets, `f_data` points to the socket structure associated with the descriptor. Finally, we see that `so_proto` in the socket structure points to the `protosw` structure for the protocol selected when the socket is created. Recall that each `protosw` structure is shared by all sockets associated with the protocol.

We now proceed to discuss the system calls.



## Chapter 15. Socket Layer

### 15.6 socket System Call

The socket system call creates a new socket and returns a descriptor. The domain, type, and protocol arguments specified in Figure 15.14) allocates a new descriptor, which identifies the descriptor to the process.

**Figure 15.14. socket System Call**

```
— uipc_syscalls.c
42 struct socket_args {
43     int      domain;
44     int      type;
45     int      protocol;
46 };
47 socket(p, uap, retval)
48 struct proc *p;
49 struct socket_args *uap;
50 int      *retval;
51 {
52     struct filedesc *fdp = p->p_fd;
53     struct socket *so;
54     struct file *fp;
55     int      fd, error;
56     if (error = falloc(p, &fp, &fd))
57         return (error);
58     fp->f_flag = FREAD | FWRITE;
59     fp->f_type = DTTYPE_SOCKET;
60     fp->f_ops = &socketops;
61     if (error = socrreate(uap->domain, &so, uap->type, uap->protocol)) {
62         fdp->fd_ofiles[fd] = 0;
63         ffree(fp);
64     } else {
65         fp->f_data = (caddr_t) so;
66         *retval = fd;
67     }
68     return (error);
69 }
```

— uipc\_syscalls.c

## 42-55

Before each system call a structure is defined to the kernel. In this case, the arguments are p (process); uap, a pointer to a structure containing the system call; and retval, a value/return argument. Normally, we ignore the p and retval arguments pointed to by uap as the arguments to the system call.

## 56-60

falloc allocates a new file structure and slot in the new structure and fd is the index of the structure for read and write access and marks it.

shared by all sockets, is attached to the file structure initialized at compile time as shown in [Figure 1](#).

**Figure 15.15. socketops: the global operations table**

Member	Value
fo_read	<i>soo_read</i>
fo_write	<i>soo_write</i>
fo_ioctl	<i>soo_ioctl</i>
fo_select	<i>soo_select</i>
fo_close	<i>soo_close</i>

60-69

`socreate` allocates and initializes a socket structure. If an error occurs, the file structure is released, and the descriptor is set to point to the socket structure and establishes the socket. `fd` is returned to the process through a pointer returned by `socreate`.

## **socreate Function**

Most socket system calls are divided into at least two functions. `socreate` are. The first function retrieves from the table of global operations the second function to do the work, and then returns the result. The second function can be called directly by kernel code. The `socreate` function is shown in [Figure 15.16](#).

## Figure 15.16. `socreate`

```
43 socreate(dom, aso, type, proto)                                uipc_socket.c
44 int      dom;
45 struct socket **aso;
46 int      type;
47 int      proto;;
48 {
49     struct proc *p = curproc; /* XXX */
50     struct protosw *prp;
51     struct socket *so;
52     int      error;
53
54     if (proto)
55         prp = pfifindproto(dom, proto, type);
56     else
57         prp = pfifindtype(dom, type);
58     if (prp == 0 || prp->pr_usrreq == 0)
59         return (EPROTONOSUPPORT);
60     if (prp->pr_type != type)
61         return (EPROTOTYPE);
62     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_WAIT);
63     bzero((caddr_t) so, sizeof(*so));
64     so->so_type = type;
65     if (p->p_ucred->cr_uid == 0)
66         so->so_state = SS_PRIV;
67     so->so_proto = prp;
68     error =
69         (*prp->pr_usrreq) (so, PRU_ATTACH,
70                             (struct mbuf *) 0, (struct mbuf *) proto, (struct mbuf *) 0);
71     if (error) {
72         so->so_state |= SS_NOFDREF;
73         soffree(so);
74     }
75     *aso = so;
76     return (0);
77 }
```

43-52

The four arguments to `socreate` are: `dom`, the domain in which a pointer to a new socket structure is returned (`so`); `type` (e.g., `SOCK_STREAM`); and `proto`, the requested protocol.

## Find protocol switch table

53-60

If proto is nonzero, pffindproto looks for the specific protocol number; if proto is zero, pffindtype looks for a protocol within the specified type. Both functions return a pointer to a protocol structure (Section 7.6).

## Allocate and initialize socket structure

61-66

socreate allocates a new socket structure, fills it with information, and if the process has superuser privileges, turns on SS\_PRIV.

## PRU\_ATTACH request

67-69

The first example of the protocol-independent socket interface appears in socreate. Recall from Section 7.4 an explanation of how a protocol module provides a pointer to the user-request function of the protocol structure. The prototype for this function in order to handle communication requests is:

```
int pr_usrreq (struct socket *so, ...);
```

The first argument, so, is a pointer to the relevant socket structure. The next three arguments (not shown) are passed as pointers to mbuf structures. They are always passed as pointers to mbuf structures.

used when necessary to avoid warnings from the compiler.

Figure 15.17 shows the requests available through the PRU interface. The arguments for each request depend on the particular protocol service being used.

**Figure 15.17. p**

Request	Arguments			Description
	m0	m1	m2	
<i>PRU_ABORT</i>				abort any existing connection
<i>PRU_ACCEPT</i>				wait for and accept a connection
<i>PRU_ATTACH</i>				a new socket has been created
<i>PRU_BIND</i>				bind the address to the socket
<i>PRU_CONNECT</i>				establish association or connection to address
<i>PRU_CONNECT2</i>				connect two sockets together
<i>PRU_DETACH</i>				socket is being closed
<i>PRU_DISCONNECT</i>				break association between socket and foreign address
<i>PRU_LISTEN</i>				begin listening for connections
<i>PRU_PEERADDR</i>				return foreign address associated with socket
<i>PRU_RCVD</i>				process has accepted some data
<i>PRU_RCVOOB</i>	buffer	flags		receive OOB data
<i>PRU_SEND</i>	data	flags		send regular data
<i>PRU_SENDOOB</i>	data	address	control	send OOB data
<i>PRU_SHUTDOWN</i>				end communication with foreign address
<i>PRU_SOCKADDR</i>		buffer		return local address associated with socket

*PRU\_CONNECT2* is supported only within the same process. Sockets can only connect to each other. Unix pipes are implemented in user space.

## Cleanup and return

70-77

Returning to `socreate`, the function attaches the socket to the PRU interface by issuing the *PRU\_ATTACH* request to notify the PRU interface of the socket's creation. It also attaches the socket to most protocols, including TCP and UDP, to allocate memory for them.

support the new end point.

## Superuser Privileges

Figure 15.18 summarizes the networking opera

Figure 15.18. Superuser privileges

Function	Superuser		Description	Reference
	Process	Socket		
in_control		•	interface address, netmask, and destination address assignment	Figure 6.14
in_control		•	broadcast address assignment	Figure 6.22
in_pcbbind			binding to an Internet port less than 1024	Figure 22.22
ifioctl	•		interface configuration changes	Figure 4.29
ifioctl	•		multicast address configuration (see text)	Figure 12.11
rip_usrreq	•		creating an ICMP, IGMP, or raw IP socket	Figure 32.10
slopen	•		associating a SLIP device with a tty device	Figure 5.9

The multicast ioctl commands (SIOCADDMULTI and SIOCDELMULTI) require superuser processes when they are invoked in the context of an IP\_DROP\_MEMBERSHIP socket options (Section 15.1).

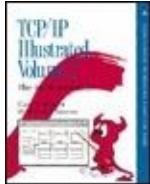
In Figure 15.18, the "Process" column identifies a process, and the "Socket" column identifies a socket. A value of "•" indicates that a superuser process (i.e., the process does not have superuser privileges) can invoke the function on a socket, [Exercise 15.1](#)). In Net/3, the suser function tests SS\_PRIV to determine if a process has superuser privileges, and the SS\_PRIV flag determines if a process is a superuser process.

Since rip\_usrreq tests SS\_PRIV immediately after the function is called, it is accessible only from a superuser process.

---

**Team-Fly**





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.7 getsock and sockargs Functions

These functions appear repeatedly in the implementation of the socket system calls. `getsock` maps a descriptor to a file table entry and `sockargs` copies arguments from the process to a newly allocated mbuf in the kernel. Both functions check for invalid arguments and return a nonzero error code accordingly.

Figure 15.19 shows the `getsock` function.

**Figure 15.19. `getsock` function.**

```
754 getsock(fd, fdes, fpp)                                uipc_syscalls.c
755 struct filedesc *fdp;
756 int     fdes;
757 struct file **fpp;
758 {
759     struct file *fp;
760     if ((unsigned) fdes >= fdp->fd_nfiles ||          uipc_syscalls.c
761         *(fp = fdp->fd_ofiles[fdes]) == NULL)
762         return (EBADF);
763     if (fp->f_type != DTTYPE_SOCKET)
764         return (ENOTSOCK);
765     *fpp = fp;
766     return (0);
767 }
```

## 754-767

The function selects the file table entry specified by the descriptor fdes with fdp, a pointer to the filedesc structure. getsock returns a pointer to the open file structure in fpp or an error if the descriptor is out of the valid range, does not point to an open file, or does not have a socket associated with it.

Figure 15.20 shows the sockargs function.

**Figure 15.20. sockargs function.**

```
768 sockargs(mp, buf, buflen, type)           uipc_syscalls.c
769 struct mbuf **mp;
770 caddr_t buf;
771 int     buflen, type;
772 {
773     struct sockaddr *sa;
774     struct mbuf *m;
775     int     error;
776     if ((u_int) buflen > MLEN) {
777         return (EINVAL);
778     }
779     m = m_get(M_WAIT, type);
780     if (m == NULL)
781         return (ENOBUFS);
782     m->m_len = buflen;
783     error = copyin(buf, mtod(m, caddr_t), (u_int) buflen);
784     if (error)
785         (void) m_free(m);
786     else {
787         *mp = m;
788         if (type == MT_SONAME) {
789             sa = mtod(m, struct sockaddr *);
790             sa->sa_len = buflen;
791         }
792     }
793     return (error);
794 }
```

uipc\_syscalls.c

## 768-783

The mechanism described in [Section 15.4](#) copies pointer arguments for a system call from the process to the kernel but does not copy the data referenced by the pointers, since the semantics of each argument are known only by the specific system call and not by the generic system call mechanism. Several system calls use `sockargs` to follow the pointer arguments and copy the referenced data from the process into a newly allocated `mbuf` within the kernel. For example, `sockargs` copies the local socket address pointed to by `bind`'s second argument from the process

to an mbuf.

If the data does not fit in a single mbuf or an mbuf cannot be allocated, sockargs returns EINVAL or ENOBUFS. Note that a standard mbuf is used and not a packet header mbuf. copyin copies the data from the process into the mbuf. The most common error from copyin is EACCES, returned when the process provides an invalid address.

## 784-785

When an error occurs, the mbuf is discarded and the error code is returned. If there is no error, a pointer to the mbuf is returned in mp, and sockargs returns 0.

## 786-794

If type is MT SONAME, the process is passing in a sockaddr structure. sockargs sets the internal length, sa\_len, to the length of the argument just copied. This ensures that the size contained within the structure is correct even if the process did not initialize the structure correctly.

Net/3 does include code to support applications compiled on a pre-4.3BSD Reno system, which did not have an `sa_len` member in the `sockaddr` structure, but that code is not shown in [Figure 15.20](#).

---

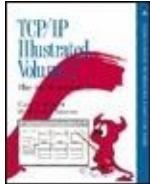
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.8 bind System Call

The bind system call associates a local network transport address with a socket. A process acting as a client usually does not care what its local address is. In this case, it isn't necessary to call bind before the process attempts to communicate; the kernel selects and implicitly binds a local address to the socket as needed.

A server process almost always needs to bind to a specific well-known address. If so, the process must call bind before accepting connections (TCP) or receiving datagrams (UDP), because the clients establish connections or send datagrams to the well-known address.

A socket's foreign address is specified by connect or by one of the write calls that allow specification of foreign addresses (sendto or sendmsg).

Figure 15.21 shows bind.

## Figure 15.21. bind function.

```
70 struct bind_args {
71     int      s;
72     caddr_t name;
73     int      namelen;
74 };
75 bind(p, uap, retval)
76 struct proc *p;
77 struct bind_args *uap;
78 int      *retval;
79 {
80     struct file *fp;
81     struct mbuf *nam;
82     int      error;
83     if (error = getsock(p->p_fd, uap->s, &fp))
84         return (error);
85     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
86         return (error);
87     error = sobind((struct socket *) fp->f_data, nam);
88     m_freem(nam);
89     return (error);
90 }
```

70-82

The arguments to bind (passed within a bind\_args structure) are: s, the socket descriptor; name, a pointer to a buffer containing the transport address (e.g., a sockaddr\_in structure); and namelen, the

size of the buffer.

83-90

getsock returns the file structure for the descriptor, and sockargs copies the local address from the process into an mbuf, sobind associates the address specified by the process with the socket. Before bind returns sobind's result, the mbuf holding the address is released.

Technically, a descriptor such as `s` identifies a file structure with an associated socket structure and is not itself a socket structure. We refer to such a descriptor as a socket to simplify our discussion.

We will see this pattern many times: arguments specified by the process are copied into an mbuf and processed as necessary, and then the mbuf is released before the system call returns. Although mbufs were designed explicitly to facilitate processing of network data packets, they are also effective as a general-purpose dynamic memory allocation mechanism.

Another pattern illustrated by bind is that retval is unused in many system calls. In [Section 15.4](#) we mentioned that retval is always initialized to 0 before syscall dispatches control to a system call. If 0 is the appropriate return value, the system calls do not need to change retval.

## sobind Function

sobind, shown in [Figure 15.22](#), is a wrapper that issues the PRU\_BIND request to the protocol associated with the socket.

**Figure 15.22. sobind function.**

```
78 sobind(so, nam)                                     uipc_socket.c
79 struct socket *so;
80 struct mbuf *nam;
81 {
82     int      s = splnet();
83     int      error;
84     error =
85         (*so->so_proto->pr_usrreq) (so, PRU_BIND,
86                                         (struct mbuf *) 0, nam, (struct mbuf *) 0);
87     splx(s);
88     return (error);
89 }
```

78-89

sobind issues the PRU\_BIND request. The

local address, nam, is associated with the socket if the request succeeds; otherwise the error code is returned.

---

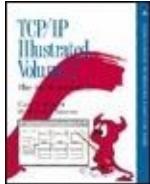
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.9 listen System Call

The listen system call, shown in [Figure 15.23](#), notifies a protocol that the process is prepared to accept incoming connections on the socket. It also specifies a limit on the number of connections that can be queued on the socket, after which the socket layer refuses to queue additional connection requests. When this occurs, TCP ignores incoming connection requests. Queued connections are made available to the process when it calls accept ([Section 15.11](#)).

**Figure 15.23. listen system call.**

```
91 struct listen_args {
92     int     s;
93     int     backlog;
94 };
95 listen(p, uap, retval)
96 struct proc *p;
97 struct listen_args *uap;
98 int     *retval;
99 {
100     struct file *fp;
101     int     error;
102     if (error = getsock(p->p_fd, uap->s, &fp))
103         return (error);
104     return (solisten((struct socket *) fp->f_data, uap->backlog));
105 }
```

## 91-98

The two arguments passed to listen specify the socket descriptor and the connection queue limit.

## 99-105

getsock returns the file structure for the descriptor, s, and solisten passes the listen request to the protocol layer.

## solisten Function

This function, shown in [Figure 15.24](#), issues the PRU\_LISTEN request and prepares the socket to receive connections.

## Figure 15.24. solisten function.

```
————— uipc_socket.c ———  
90 solisten(so, backlog)  
91 struct socket *so;  
92 int      backlog;  
93 {  
94     int      s = splnet(), error;  
95     error =  
96         (*so->so_proto->pr_usrreq) (so, PRU_LISTEN,  
97             (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0);  
98     if (error) {  
99         splx(s);  
100        return (error);  
101    }  
102    if (so->so_q == 0)  
103        so->so_options |= SO_ACCEPTCONN;  
104    if (backlog < 0)  
105        backlog = 0;  
106    so->so_qlimit = min(backlog, SOMAXCONN);  
107    splx(s);  
108    return (0);  
109 }
```

————— uipc\_socket.c ———

90-109

After solisten issues the PRU\_LISTEN request and pr\_usrreq returns, the socket is marked as ready to accept connections. SS\_ACCEPTCONN is not set if a connection is queued when pr\_usrreq returns.

The maximum queue size for incoming connections is computed and saved in so\_qlimit. Here Net/3 silently enforces a lower limit of 0 and an upper limit of 5 (SOMAXCONN) backlogged connections.

Top

## Chapter 15. Socket Layer

---

### 15.10 tsleep and wakeup Functions

When a process executing within the kernel calls the `tsleep` function and the resource is unavailable, it waits for the resource to become available. The `tsleep` function has the following prototype:

```
int tsleep (caddr_t chan, int pri, char *msg, int sec);
```

The first argument to `tsleep`, *chan*, is called the channel or event on which the process is waiting. Many processes can be waiting on the same channel. When the resource becomes available or when another process wakes up the process with the wait channel as the single argument to the `wakeup` function, the process is:

```
void wakeup (caddr_t chan);
```

All processes waiting for the channel are awakened. The kernel arranges for `tsleep` to return when each process is woken up.

execution.

The *pri* argument specifies the priority of the process as well as several optional control flags for tsleep. tsleep also returns when a signal arrives, *mesg*. *tsleep* and is included in debugging messages as a upper bound on the sleep period and is measured.

Figure 15.25 summarizes the return values from

**Figure 15.25. tsleep return values**

<code>tsleep()</code>	Description
0	The process was awakened by a matching call to <code>wakeup</code> .
<code>EWOULDBLOCK</code>	The process was awakened after sleeping for <i>timeo</i> clock ticks and before the matching call to <code>wakeup</code> .
<code>ERESTART</code>	A signal was handled by the process during the sleep and the pending system call should be restarted.
<code>EINTR</code>	A signal was handled by the process during the sleep and the pending system call should fail.

A process never sees the `ERESTART` error because it function and never returned to a process.

Because all processes sleeping on a wait channel always see a call to tsleep within a tight loop. Even if the resource is available before proceeding because another process may have claimed the resource first. If the resource becomes available again, the process calls tsleep once again.

It is unusual for multiple processes to be sleeping

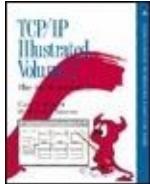
wakeup usually causes only one process to be woken up.

For a more detailed discussion of the sleep and wakeup mechanisms [Berg et al. 1989].

## Example

One use of multiple processes sleeping on the same socket is in multiple server processes reading from a UDP socket. In fact, as long as no data is available, the calls blocking on a socket until data arrives on the socket, the socket layer calls wake up the processes in the run queue. The first server to run receives the data and can then tsleep again. In this way, incoming datagrams can be handled without the cost of starting a new process for each connection. This technique can also be used to process incoming connection requests by having multiple processes call accept on the same socket. This is described in [Berg and Stevens 1993].

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.11 accept System Call

After calling `listen`, a process waits for incoming connections by calling `accept`, which returns a descriptor that references a new socket connected to a client. The original socket, `s`, remains unconnected and ready to receive additional connections. `accept` returns the address of the foreign system if `name` points to a valid buffer.

The connection-processing details are handled by the protocol associated with the socket. For TCP, the socket layer is notified when a connection has been established (i.e., when TCP's three-way handshake has completed). For other

protocols, such as OSI's TP4, tsleep returns when a connection request has arrived. The connection is completed when explicitly confirmed by the process by reading or writing on the socket.

[Figure 15.26](#) shows the implementation of accept.

**Figure 15.26. accept system call.**

```
106 struct accept_args {
107     int      s;
108     caddr_t name;
109     int      *namelen;
110 };
111 accept(p, uap, retval)
112 struct proc *p;
113 struct accept_args *uap;
114 int      *retval;
115 {
116     struct file *fp;
117     struct mbuf *nam;
118     int      namelen, error, s;
119     struct socket *so;
120
121     if (uap->name && (error = copyin((caddr_t) uap->namelen,
122                                         (caddr_t) & namelen, sizeof(namelen)))) {
123         return (error);
124     }
125     if (error = getsock(p->p_fd, uap->s, &fp))
126         return (error);
127     s = splnet();
128     so = (struct socket *) fp->f_data;
129     if ((so->so_options & SO_ACCEPTCONN) == 0) {
130         splx(s);
131         return (EINVAL);
132     }
133     if ((so->so_state & SS_NBIO) && so->so_qlen == 0) {
134         splx(s);
135         return (EWOULDBLOCK);
136     }
137     while (so->so_qlen == 0 && so->so_error == 0) {
138         if (so->so_state & SS_CANTRCVMORE) {
139             so->so_error = ECONNABORTED;
140             break;
141         }
142         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
143                            netcon, 0)) {
144             splx(s);
145             return (error);
146         }
147         if (so->so_error) {
148             error = so->so_error;
149             so->so_error = 0;
150             splx(s);
151             return (error);
152         }
153         if (error = fallloc(p, &fp, retval)) {
154             splx(s);
155             return (error);
156         }
157     }
158 }
```

```

156     { struct socket *aso = so->so_q;
157     if (soqremque(aso, 1) == 0)
158         panic("accept");
159     so = aso;
160 }
161 fp->f_type = DTTYPE_SOCKET;
162 fp->f_flag = FREAD | FWRITE;
163 fp->f_ops = &socketops;
164 fp->f_data = (caddr_t) so;
165 nam = m_get(M_WAIT, MT_SONAME);
166 (void) soaccept(so, nam);
167 if (uap->name) {
168     if (namelen > nam->m_len)
169         namelen = nam->m_len;
170     /* SHOULD COPY OUT A CHAIN HERE */
171     if ((error = copyout(mtod(nam, caddr_t), (caddr_t) uap->name,
172                         (u_int) namelen)) == 0)
173         error = copyout((caddr_t) &namelen,
174                         (caddr_t) uap->anamelen, sizeof(*uap->anamelen));
175 }
176 m_freem(nam);
177 splx(s);
178 return (error);
179 }

```

*uipc\_syscalls.c*

## 106-114

The three arguments to accept (in the accept\_args structure) are: s, the socket descriptor; name, a pointer to a buffer to be filled in by accept with the transport address of the foreign host; and anamelen, a pointer to the size of the buffer.

## Validate arguments

## 116-134

accept copies the size of the buffer (\*anamelen) into namelen, and getsock

returns the file structure for the socket. If the socket is not ready to accept connections (i.e., listen has not been called) or nonblocking I/O has been requested and no connections are queued, EINVAL or EWOULDBLOCK are returned respectively.

## Wait for a connection

135-145

The while loop continues until a connection is available, an error occurs, or the socket can no longer receive data. accept is not automatically restarted after a signal is caught (tsleep returns EINTR). The protocol layer wakes up the process when it inserts a new connection on the queue with sonewconn.

Within the loop, the process waits in tsleep, which returns 0 when a connection is available. If tsleep is interrupted by a signal or the socket is set for nonblocking semantics, accept returns EINTR or EWOULDBLOCK ([Figure 15.25](#)).

## Asynchronous errors

146-151

If an error occurred on the socket during the sleep, the error code is moved from the socket to the return value for accept, the socket error is cleared, and accept returns.

It is common for asynchronous events to change the state of a socket. The protocol processing layer notifies the socket layer of the change by setting `so_error` and waking any process waiting on the socket. Because of this, the socket layer must always examine `so_error` after waking to see if an error occurred while the process was sleeping.

## Associate socket with descriptor

152-164

`falloc` allocates a descriptor for the new connection; the socket is removed from the accept queue by `soqremque` and

attached to the file structure. [Exercise 15.4](#) discusses the call to panic.

## Protocol processing

167-179

accept allocates a new mbuf to hold the foreign address and calls soaccept to do protocol processing. The allocation and queueing of new sockets created during connection processing is described in [Section 15.12](#). If the process provided a buffer to receive the foreign address, copyout copies the address from nam and the length from nameLEN to the process. If necessary, copyout silently truncates the name to fit in the process's buffer. Finally, the mbuf is released, protocol processing enabled, and accept returns.

Because only one mbuf is allocated for the foreign address, transport addresses must fit in one mbuf. Unix domain addresses, which are pathnames in the filesystem (up to 1023 bytes in length), may encounter this limit, but there is no problem with the

16-byte sockaddr\_in structure for the Internet domain. The comment on line 170 indicates that this limitation could be removed by allocating and copying an mbuf chain.

## soaccept Function

soaccept, shown in [Figure 15.27](#), calls the protocol layer to retrieve the client's address for the new connection.

**Figure 15.27. soaccept function.**

```
184 soaccept(so, nam)                                     uipc_socket.c
185 struct socket *so;
186 struct mbuf *nam;
187 {
188     int      s = splnet();
189     int      error;
190     if ((so->so_state & SS_NOFDREF) == 0)
191         panic("soaccept: !NOFDREF");
192     so->so_state &= ~SS_NOFDREF;
193     error = (*so->so_proto->pr_usrreq) (so, PRU_ACCEPT,
194                                         (struct mbuf *) 0, nam, (struct mbuf *) 0);
195     splx(s);
196     return (error);
197 }
```

184-197

soaccept ensures that the socket is associated with a descriptor and issues the

PRU\_ACCEPT request to the protocol. After pr\_usrreq returns, nam contains the name of the foreign socket.

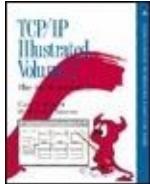
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



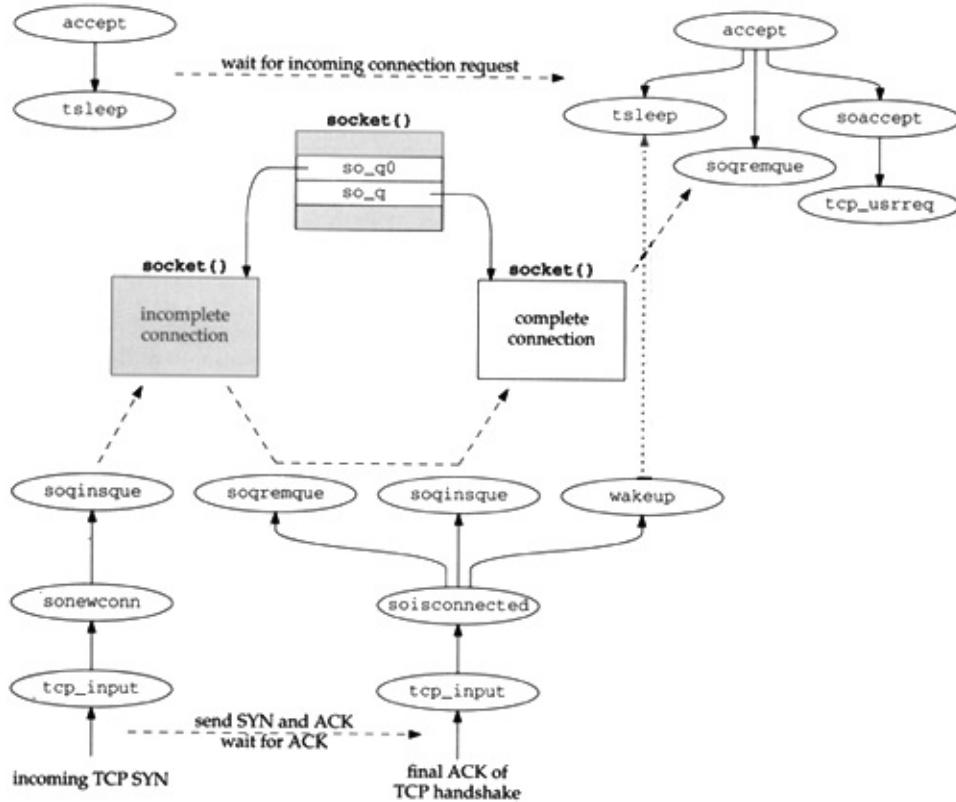
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.12 sonewconn and soisconnected Functions

In [Figure 15.26](#) we saw that accept waits for the protocol layer to process incoming connection requests and to make them available through so\_q. [Figure 15.28](#) uses TCP to illustrate this process.

**Figure 15.28. Incoming TCP connection processing.**



In the upper left corner of [Figure 15.28](#), `accept` calls `tsleep` to wait for incoming connections. In the lower left, `tcp_input` processes an incoming TCP SYN by calling `sonewconn` to create a socket for the new connection ([Figure 28.7](#)). `sonewconn` queues the socket on `so_q0`, since the three-way handshake is not yet complete.

When the final ACK of the TCP handshake arrives, `tcp_input` calls `soisconnected` ([Figure 29.2](#)), which updates the new socket, moves it from `so_q0` to `so_q`, and wakes up any processes that had called

accept to wait for incoming connections.

The upper right corner of the figure shows the functions we described with [Figure 15.26](#). When tsleep returns, accept takes the connection off so\_q and issues the PRU\_ATTACH request. The socket is associated with a new file descriptor and returned to the calling process.

[Figure 15.29](#) shows the sonewconn function.

### **Figure 15.29. sonewconn function.**

---

```

123 struct socket *
124 sonewconn(head, connstatus)
125 struct socket *head;
126 int connstatus;
127 {
128     struct socket *so;
129     int soqueue = connstatus ? 1 : 0;
130     if (head->so_qlen + head->so_qolen > 3 * head->so_qlimit / 2)
131         return ((struct socket *) 0);
132     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_DONTWAIT);
133     if (so == NULL)
134         return ((struct socket *) 0);
135     bzero((caddr_t) so, sizeof(*so));
136     so->so_type = head->so_type;
137     so->so_options = head->so_options & ~SO_ACCEPTCONN;
138     so->so_linger = head->so_linger;
139     so->so_state = head->so_state | SS_NOFDREF;
140     so->so_proto = head->so_proto;
141     so->so_timeo = head->so_timeo;
142     so->so_pgid = head->so_pgid;
143     (void) soreserve(so, head->so_snd.sb_hiwat, head->so_rcv.sb_hiwat);
144     sqinsque(head, so, soqueue);
145     if ((*so->so_proto->pr_usrreq) (so, PRU_ATTACH,
146         (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0)) {
147         (void) sqremque(so, soqueue);
148         (void) free((caddr_t) so, M_SOCKET);
149         return ((struct socket *) 0);
150     }
151     if (connstatus) {
152         sorwakeup(head);
153         wakeup((caddr_t) & head->so_timeo);
154         so->so_state |= connstatus;
155     }
156     return (so);
157 }

```

---

uipc\_socket2.c

## 123-129

The protocol layer passes head, a pointer to the socket that is accepting the incoming connection, and connstatus, a flag to indicate the state of the new connection. For TCP, connstatus is always 0.

For TP4, connstatus is always SS\_ISCONFIRMING. The connection is implicitly confirmed when a process

begins reading from or writing to the socket.

## Limit incoming connections

130-131

sonewconn prohibits additional connections when the following inequality is true:

$$\text{so_qlen} + \text{so_q0len} > \frac{3 \times \text{so_qlimit}}{2}$$

This formula provides a fudge factor for connections that never complete and guarantees that listen (fd, 0) allows one connection. See Figure 18.23 in Volume 1 for an additional discussion of this formula.

## Allocate new socket

132-143

A new socket structure is allocated and initialized. If the process calls setsockopt

for the listening socket, the connected socket inherits several socket options because `so_options`, `so_linger`, `so_pgid`, and the `sb_hiwat` values are copied into the new socket structure.

## Queue connection

144

`soqueue` was set from `connstatus` on line 129. The new socket is inserted onto `so_q0` if `soqueue` is 0 (e.g., TCP connections) or onto `so_q` if `connstatus` is nonzero (e.g., TP4 connections).

## Protocol processing

145-150

The `PRU_ATTACH` request is issued to perform protocol layer processing on the new connection. If this fails, the socket is dequeued and discarded, and `sonewconn` returns a null pointer.

## Wakeup processes

151-157

If connstatus is nonzero, any processes sleeping in accept or selecting for readability on the socket are awakened, connstatus is logically ORed with so\_state. This code is never executed for TCP connections, since connstatus is always 0 for TCP.

Protocols, such as TCP, that put incoming connections on so\_q0 first, call soisconnected when the connection establishment phase completes. For TCP, this happens when the second SYN is ACKed on the connection.

Figure 15.30 shows soisconnected.

**Figure 15.30. soisconnected function.**

```
78 soisconnected(so)
79 struct socket *so;
80 {
81     struct socket *head = so->so_head;
82     so->so_state &= ~(SS_ISCONNECTING | SS_ISDISCONNECTING | SS_ISCONFIRMING);
83     so->so_state |= SS_ISCONNECTED;
84     if (head && soqremque(so, 0)) {
85         soqinsque(head, so, 1);
86         sorwakeups(head);
87         wakeup((caddr_t) & head->so_timeo);
88     } else {
89         wakeup((caddr_t) & so->so_timeo);
90         sorwakeups(so);
91         sowakeups(so);
92     }
93 }
```

uipc\_socket2.c

## Queue incomplete connections

78-87

The socket state is changed to show that the connection has completed. When soisconnected is called for incoming connections, (i.e., when the local process is calling accept), head is nonnull.

If soqremque returns 1, the socket is queued on so\_q and sorwakeups wakes up any processes using select to monitor the socket for connection arrival by testing for readability. If a process is blocked in accept waiting for the connection, wakeup causes the matching tsleep to return.

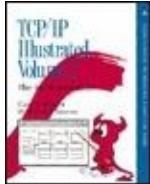
## Wakeup processes waiting for new

## connection

88-93

If head is null, soqremque is not called since the process initiated the connection with the connect system call and the socket is not on a queue. If head is nonnull and soqremque returns 0, the socket is already on so\_q. This happens with protocols such as TP4, which place connections on so\_q before they are complete. wakeup awakens any process blocked in connect, and sorwakeup and sowakeup take care of any processes that are using select to wait for the connection to complete.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.13 connect System call

A server process calls the listen and accept system calls to wait for a remote process to initiate a connection. If the process wants to initiate a connection itself (i.e., a client), it calls connect.

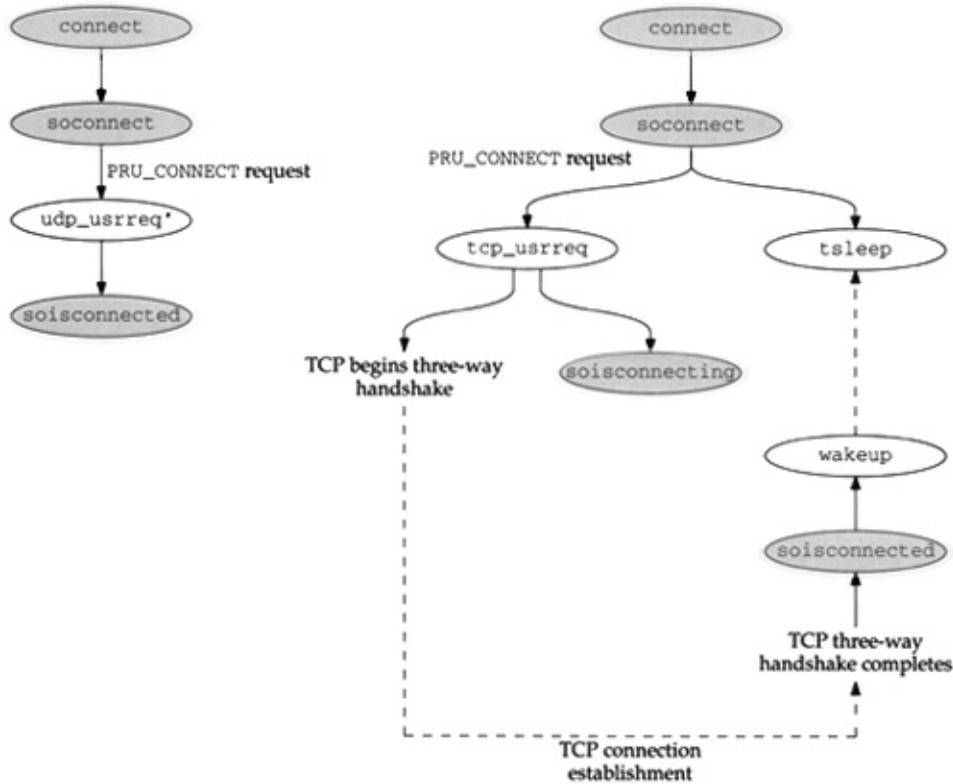
For connection-oriented protocols such as TCP, connect establishes a connection to the specified foreign address. The kernel selects and implicitly binds an address to the local socket if the process has not already done so with bind.

For connectionless protocols such as UDP or ICMP, connect records the foreign address for use in sending future

datagrams. Any previous foreign address is replaced with the new address.

Figure 15.31 shows the functions called when connect is used for UDP or TCP.

**Figure 15.31. connect processing.**



The left side of the figure shows connect processing for connectionless protocols, such as UDP. In this case the protocol layer calls **soisconnected** and the connect system call returns immediately.

The right side of the figure shows connect processing for connection-oriented protocols, such as TCP. In this case, the protocol layer begins the connection establishment and calls `soisconnecting` to indicate that the connection will complete some time in the future. Unless the socket is nonblocking, `soconnect` calls `tsleep` to wait for the connection to complete. For TCP, when the three-way handshake is complete, the protocol layer calls `soisconnected` to mark the socket as connected and then calls `wakeup` to awaken the process and complete the connect system call.

Figure 15.32 shows the connect system call.

### **Figure 15.32. connect system call.**

```

180 struct connect_args {
181     int      s;
182     caddr_t name;
183     int      namelen;
184 };
185 connect(p, uap, retval)
186 struct proc *p;
187 struct connect_args *uap;
188 int    *retval;
189 {
190     struct file *fp;
191     struct socket *so;
192     struct mbuf *nam;
193     int      error, s;
194     if (error = getsock(p->p_fd, uap->s, &fp))
195         return (error);
196     so = (struct socket *) fp->f_data;
197     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING))
198         return (EALREADY);
199     if (error = sockargs(&nam, uap->name, uap->namelen, MT_SONAME))
200         return (error);
201     error = soconnect(so, nam);
202     if (error)
203         goto bad;
204     if ((so->so_state & SS_NBIO) && (so->so_state & SS_ISCONNECTING)) {
205         m_freem(nam);
206         return (EINPROGRESS);
207     }
208     s = splnet();
209     while ((so->so_state & SS_ISCONNECTING) && so->so_error == 0)
210         if (error = tsleep((caddr_t) & so->so_timeo, PSOCK | PCATCH,
211                             netcon, 0))
212             break;
213     if (error == 0) {
214         error = so->so_error;
215         so->so_error = 0;
216     }
217     splx(s);
218 bad:
219     so->so_state &= ~SS_ISCONNECTING;
220     m_freem(nam);
221     if (error == ERESTART)
222         error = EINTR;
223     return (error);
224 }
```

## 180-188

The three arguments to connect (in the connect\_args structure) are: s, the socket descriptor; name, a pointer to a buffer containing the foreign address; and namelen, the length of the buffer.

189-200

getsock returns the socket as usual. A connection request may already be pending on a nonblocking socket, in which case EALREADY is returned. sockargs copies the foreign address from the process into the kernel.

## Start connection processing

201-208

The connection attempt is started by calling soconnect. If soconnect reports an error, connect jumps to bad. If a connection has not yet completed by the time soconnect returns and nonblocking I/O is enabled, EINPROGRESS is returned immediately to avoid waiting for the connection to complete. Since connection establishment normally involves exchanging several packets with the remote system, it may take a while to complete. Further calls to connect return EALREADY until the connection completes. EISCONN is returned when the connection

is complete.

## Wait for connection establishment

208-217

The while loop continues until the connection is established or an error occurs. splnet prevents connect from missing a wakeup between testing the state of the socket and the call to tsleep. After the loop, error contains 0, the error code from tsleep, or the error from the socket.

218-224

The SS\_ISCONNECTING flag is cleared since the connection has completed or the attempt has failed. The mbuf containing the foreign address is released and any error is returned.

## soconnect Function

This function ensures that the socket is in a valid state for a connection request. If

the socket is not connected or a connection is not pending, then the connection request is always valid. If the socket is already connected or a connection is pending, the new connection request is rejected for connection-oriented protocols such as TCP. For connectionless protocols such as UDP, multiple connection requests are OK but each new request replaces the previous foreign address.

Figure 15.33 shows the soconnect function.

### Figure 15.33. soconnect function.

```
198 soconnect(so, nam)
199 struct socket *so;
200 struct mbuf *nam;
201 {
202     int      s;
203     int      error;
204     if (so->so_options & SO_ACCEPTCONN)
205         return (EOPNOTSUPP);
206     s = splnet();
207     /*
208      * If protocol is connection-based, can only connect once.
209      * Otherwise, if connected, try to disconnect first.
210      * This allows user to disconnect by connecting to, e.g.,
211      * a null address.
212     */
213     if (so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING) &&
214         ((so->so_proto->pr_flags & PR_CONNREQUIRED) ||
215          (error = sodisconnect(so))))
216         error = EISCONN;
217     else
218         error = (*so->so_proto->pr_usrreq) (so, PRU_CONNECT,
219                                         (struct mbuf *) 0, nam, (struct mbuf *) 0);
220     splx(s);
221     return (error);
222 }
```

198-222

soconnect returns EOPNOTSUPP if the socket is marked to accept connections, since a process cannot initiate connections if listen has already been called for the socket. EISCONN is returned if the protocol is connection oriented and a connection has already been initiated. For a connectionless protocol, any existing association with a foreign address is broken by sodisconnect.

The PRU\_CONNECT request starts the appropriate protocol processing to establish the connection or the association.

## Breaking a Connectionless Association

For connectionless protocols, the foreign address associated with a socket can be discarded by calling connect with an invalid name such as a pointer to a structure filled with 0s or a structure with an invalid size. sodisconnect removes a foreign address associated with the socket, and PRU\_CONNECT returns an error such

as EAFNOSUPPORT or EADDRNOTAVAIL, leaving the socket with no foreign address. This is a useful, although obscure, way of breaking the association between a connectionless socket and a foreign address without replacing it.

---

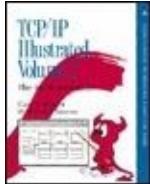
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.14 shutdown System Call

The shutdown system call, shown in [Figure 15.34](#), closes the write-half, read-half, or both halves of a connection. For the read-half, shutdown discards any data the process hasn't yet read and any data that arrives after the call to shutdown. For the write-half, shutdown lets the protocol specify the semantics. For TCP, any remaining data will be sent followed by a FIN. This is TCP's half-close feature (Section 18.5 of Volume 1).

**Figure 15.34. shutdown system call.**

```
550 struct shutdown_args {  
551     int     s;  
552     int     how;  
553 };  
554 shutdown(p, uap, retval)  
555 struct proc *p;  
556 struct shutdown_args *uap;  
557 int     *retval;  
558 {  
559     struct file *fp;  
560     int     error;  
561     if (error = getsock(p->p_fd, uap->s, &fp))  
562         return (error);  
563     return (soshutdown((struct socket *) fp->f_data, uap->how));  
564 }
```

uipc\_syscalls.c

To destroy the socket and release the descriptor, close must be called. close can also be called directly without first calling shutdown. As with all descriptors, close is called by the kernel for sockets that have not been closed when a process terminates.

550-557

In the shutdown\_args structure, s is the socket descriptor and how specifies which halves of the connection are to be closed. [Figure 15.35](#) shows the expected values for how and how++ (which is used in [Figure 15.36](#)).

**Figure 15.35. shutdown system call options.**

how	how++	Description
0	FREAD	shut down the read-half of the connection
1	FWRITE	shut down the write-half of the connection
2	FREAD/FWRITE	shut down both halves of the connection

## Figure 15.36. soshutdown function.

---

```

720 soshutdown(so, how)           uipc_socket.c
721 struct socket *so;
722 int      how;
723 {
724     struct protosw *pr = so->so_proto;
725     how++;
726     if (how & FREAD)
727         sorflush(so);
728     if (how & FWRITE)
729         return ((*pr->pr_usrreq) (so, PRU_SHUTDOWN,
730             (struct mbuf *) 0, (struct mbuf *) 0, (struct mbuf *) 0));
731     return (0);
732 }
```

---

Notice that there is an implicit numerical relationship between how and the constants FREAD and FWRITE.

558-564

shutdown is a wrapper function for soshutdown. The socket associated with the descriptor is returned by getsock, soshutdown is called, and its value is returned.

## soshutdown and sorflush Functions

The shut down of the read-half of a connection is handled in the socket layer by sorflush, and the shut down of the write-half of a connection is processed by the PRU\_SHUTDOWN request in the protocol layer. The soshutdown function is shown in [Figure 15.36](#).

720-732

If the read-half of the socket is being closed, sorflush, shown in [Figure 15.37](#), discards the data in the socket's receive buffer and disables the read-half of the connection. If the write-half of the socket is being closed, the PRU\_SHUTDOWN request is issued to the protocol.

**Figure 15.37. sorflush function.**

```
733 sorflush(so)
734 struct socket *so;
735 {
736     struct sockbuf *sb = &so->so_rcv;
737     struct protosw *pr = so->so_proto;
738     int     s;
739     struct sockbuf asb;

740     sb->sb_flags |= SB_NOINTR;
741     (void) sblock(sb, M_WAITOK);
742     s = splimp();
743     socantrcvmore(so);
744     sbunlock(sb);
745     asb = *sb;
746     bzero((caddr_t) sb, sizeof(*sb));
747     splx(s);

748     if (pr->pr_flags & PR_RIGHTS && pr->pr_domain->dom_dispose)
749         (*pr->pr_domain->dom_dispose) (asb.sb_mb);
750     sbrelease(&asb);
751 }
```

————— uipc\_socket.c

## 733-747

The process waits for a lock on the receive buffer. Because of SB\_NOINTR, sblock does not return when an interrupt occurs. splimp blocks network interrupts and protocol processing while the socket is modified, since the receive buffer may be accessed by the protocol layer as it processes incoming packets.

socantrcvmore marks the socket to reject incoming packets. A copy of the sockbuf structure is saved in asb to be used after interrupts are restored by splx. The original sockbuf structure is cleared by bzero, so that the receive queue appears to be empty.

## Release control mbufs

748-751

Some kernel resources may be referenced by control information present in the receive queue when shutdown was called. The mbuf chain is still available through sb\_mb in the copy of the sockbuf structure.

If the protocol supports access rights and has registered a dom\_dispose function, it is called here to release these resources.

In the Unix domain it is possible to pass descriptors between processes with control messages. These messages contain pointers to reference counted data structures. The dom\_dispose function takes care of discarding the references and the data structures if necessary to avoid creating an unreferenced structure and introducing a memory leak in the kernel. For more information on passing file descriptors within the Unix domain, see [[Stevens 1990](#)] and [[Leffler et al. 1989](#)].

Any input data pending when shutdown is called is discarded when sbrelease releases any mbufs on the receive queue.

Notice that the shut down of the read-half of the connection is processed entirely by the socket layer ([Exercise 15.6](#)) and the shut down of the write-half of the connection is handled by the protocol through the PRU\_SHUTDOWN request. TCP responds to the PRU\_SHUTDOWN by sending all queued data and then a FIN to close the write-half of the TCP connection.

[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.15 close System Call

The close system call works with any type of descriptor. When fd is the last descriptor that references the object, the object-specific close is called:

```
error = (*fp->f_ops->fo_close) (f
```

As shown in [Figure 15.13](#), fp->f\_ops->fo\_close socket is the function `soo_close`.

#### **soo\_close Function**

This function, shown in [Figure 15.38](#), is a wrapper for the soclose function.

## Figure 15.38. `soo_close` function.

```
152 soo_close(fp, p)                                sys_socket.c
153 struct file *fp;
154 struct proc *p;
155 {
156     int      error = 0;
157     if (fp->f_data)
158         error = soclose((struct socket *) fp->f_data);
159     fp->f_data = 0;
160     return (error);
161 }
```

sys\_socket.c

152-161

If a socket structure is associated with the file structure, `soclose` is called, `f_data` is cleared, a posted error is returned.

## `soclose` Function

This function aborts any connections that are pending on the socket (i.e., that have not yet been accepted by a process), waits for data to be transmitted to the foreign system, and releases the data structures that are no longer needed.

`soclose` is shown in Figure 15.39.

## Figure 15.39. `soclose` function.

```

129 soclose(so)
130 struct socket *so;
131 {
132     int      s = splnet();           /* conservative */
133     int      error = 0;
134
135     if (so->so_options & SO_ACCEPTCONN) {
136         while (so->so_q0)
137             (void) soabort(so->so_q0);
138         while (so->so_q)
139             (void) soabort(so->so_q);
140     }
141     if (so->so_pcb == 0)
142         goto discard;
143     if (so->so_state & SS_ISCONNECTED) {
144         if ((so->so_state & SS_ISDISCONNECTING) == 0) {
145             error = sodisconnect(so);
146             if (error)
147                 goto drop;
148         }
149         if (so->so_options & SO_LINGER) {
150             if ((so->so_state & SS_ISDISCONNECTING) &&
151                 (so->so_state & SS_NBIO))
152                 goto drop;
153             while (so->so_state & SS_ISCONNECTED)
154                 if (error = tsleep((caddr_t) & so->so_timeo,
155                                     PSOCK | PCATCH, netcls, so->so_linger))
156                     break;
157         }
158     drop:
159         if (so->so_pcb) {
160             int      error2 =
161             (*so->so_proto->pr_usrreq) (so, PRU_DETACH,
162                                         (struct mbuf *) 0, (struct mbuf *) 0,
163                                         (struct mbuf *) 0);
164             if (error == 0)
165                 error = error2;
166         }
167     discard:
168         if (so->so_state & SS_NOFDREP)
169             panic("soclose: NOFDREP");
170         so->so_state |= SS_NOFDREP;
171         splx(s);
172         return (error);
173 }

```

uipc\_socket.c

## Discard pending connections

129-141

If the socket was accepting connections, soclose traverses the two connection queues and calls soabort for each pending connection. If the protocol control block is null, the protocol has already been detached.

from the socket and soclose jumps to the clean at discard.

soabort issues the PRU\_ABORT request to the socket's protocol and returns the result. soabot not shown in this text. [Figures 23.38](#) and [30.1](#) discuss how UDP and TCP handle this request

## Break established connection or association

142-157

If the socket is not connected, execution continues to drop; otherwise the socket must be disconnected from its peer. If a disconnect is not in progress, sodev starts the disconnection process. If the SO\_LINGER socket option is set, soclose may need to wait for the disconnect to complete before returning. A non-blocking socket never waits for a disconnect to complete; instead, soclose jumps immediately to drop in that case. Otherwise, the connection termination is in progress, and the SO\_LINGER option indicates that soclose will wait some time for it to complete. The while loop continues until the disconnect completes, the linger time (so\_linger) expires, or a signal is delivered to the process.

If the linger time is set to 0, tsleep returns on when the disconnect completes (perhaps because of an error) or a signal is delivered.

## Release data structures

158-173

If the socket still has an attached protocol, the PRU\_DETACH request breaks the connection between this socket and the protocol. Finally the socket is marked as not having an associated file descriptor which allows sofree to release the socket.

The sofree function is shown in [Figure 15.40](#).

**Figure 15.40. sofree function.**

```
uipc_socket.c
110 sofree(so)
111 struct socket *so;
112 {
113     if (so->so_pcb || (so->so_state & SS_NOFDREF) == 0)
114         return;
115     if (so->so_head) {
116         if (!soqremque(so, 0) && !soqremque(so, 1))
117             panic("sofree dq");
118         so->so_head = 0;
119     }
120     sbrelease(&so->so_snd);
121     sorflush(so);
122     FREE(so, M_SOCKET);
123 }
```

uipc\_socket.c

## **Return if socket still in use**

**110-114**

If a protocol is still associated with the socket, socket is still associated with a descriptor, sofre returns immediately.

## **Remove from connection queues**

**115-119**

If the socket is on a connection queue (so\_head nonnull), soqremque is called to remove the so attempt is made to remove the socket from the incomplete connection queue and if this fails, t from the completed connection queue. One of t removals must succeed or the kernel panics, si so\_head was nonnull. so\_head is cleared.

## **Discard send and receive queues**

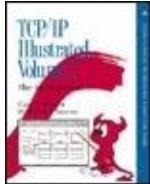
**120-123**

sbrelease discards any buffers in the send queue sorflush discards any buffers in the receive queue Finally, the socket itself is released.

---

**Team-Fly**





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 15. Socket Layer

### 15.16 Summary

In this chapter we looked at all the system calls related to network operations. The system call mechanism was described, and we traced the calls until they entered the protocol processing layer through the `pr_usrreq` function.

While looking at the socket layer, we avoided any discussion of address formats, protocol semantics, or protocol implementations. In the upcoming chapters we tie together the link-layer processing and socket-layer processing by looking in detail at the implementation of the Internet protocols in the protocol processing layer.

## Exercises

How can a process *without* superuser  
**15.1** privileges gain access to a socket  
created by a superuser process?

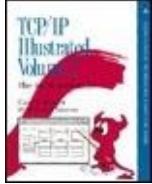
How can a process determine if the  
**15.2** sockaddr buffer it provides to accept  
was too small to hold the foreign  
address returned by the call?

A feature proposed for IPv6 sockets  
is to have accept and recvfrom  
return a source route as an array of  
128-bit IPv6 addresses instead of a  
single peer address. Since the array  
**15.3** will not fit in a single mbuf, modify  
accept and recvfrom to handle an  
mbuf chain from the protocol layer  
instead of a single mbuf. Will the  
existing code work if the protocol  
layer returns the array in an mbuf  
cluster instead of a chain of mbufs?

**15.4** Why is panic called when sqremque returns a null pointer in Figure 15.26?

**15.5** Why does sorflush make a copy of the receive buffer?

What happens when additional data is received after sorflush has zeroed  
**15.6** the socket's receive buffer? Read Chapter 16 before attempting this exercise.



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 16. Socket I/O

[Section 16.1. Introduction](#)

[Section 16.2. Code Introduction](#)

[Section 16.3. Socket Buffers](#)

[Section 16.4. write, writev, sendto, and  
sendmsg System Calls](#)

[Section 16.5. sendmsg System Call](#)

[Section 16.6. sendit Function](#)

[Section 16.7. sosend Function](#)

[Section 16.8. read, readv, recvfrom,  
and recvmsg System Calls](#)

[Section 16.9. recvmsg System Call](#)

[Section 16.10. recvit Function](#)

[Section 16.11. soreceive Function](#)

[Section 16.12. soreceive Code](#)

## Section 16.13. select System Call

## Section 16.14. Summary

---

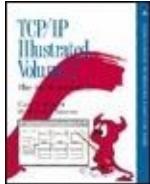
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.1 Introduction

In this chapter we discuss the system calls that read and write data on a network connection. The chapter is divided into three parts.

The first part covers the four system calls for sending data: write, writev, sendto, and sendmsg. The second part covers the four system calls for receiving data: read, readv, recvfrom, and recvmsg. The third part of the chapter covers the select system call, which provides a standard way to monitor the status of descriptors in general and sockets in particular.

The core of the socket layer is the sosend

and `soreceive` functions. They handle all I/O between the socket layer and the protocol layer. As we'll see, the semantics of the various types of protocols overlap in these functions, making the functions long and complex.

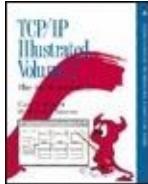
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

## 16.2 Code Introduction

The three headers and four C files listed in [Figure 16.1](#) are covered in this chapter.

### Figure 16.1. Files discussed in this chapter.

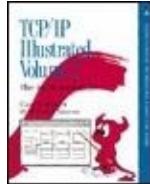
File	Description
sys/socket.h	structures and macro for sockets API
sys/socketvar.h	socket structure and macros
sys/uio.h	uio structure definition
kern/uipc_syscalls.c	socket system calls
kern/uipc_socket.c	socket layer processing
kern/sys_generic.c	select system call
kern/sys_socket.c	select processing for sockets

## Global Variables

The first two global variables shown in [Figure 16.2](#) are used by the select system call. The third global variable controls the amount of memory allocated to a socket.

## Figure 16.2. Global variables introduced in this chapter.

Variable	Datatype	Description
selwait	int	wait channel for select
nsecoll	int	flag used to avoid race conditions in select
sb_max	u_long	maximum number of bytes to allocate for a socket receive or send buffer



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.3 Socket Buffers

Section 15.3 showed that each socket has an associated send and receive buffer. The sockbuf structure definition from Figure 15.5 is repeated in Figure 16.3.

**Figure 16.3. sockbuf structure.**

```
72     struct sockbuf {
73         u_long    sb_cc;           /* actual chars in buffer */
74         u_long    sb_hiwat;        /* max actual char count */
75         u_long    sb_mbcnt;        /* chars of mbufs used */
76         u_long    sb_mbmax;        /* max chars of mbufs to use */
77         long      sb_lowat;        /* low water mark */
78         struct mbuf *sb_mb;        /* the mbuf chain */
79         struct selinfo sb_sel;    /* process selecting read/write */
80         short     sb_flags;        /* Figure 16.5 */
81         short     sb_timeo;        /* timeout for read/write */
82     } so_rcv, so_snd;
```

Each buffer contains control information as well as pointers to data stored in mbuf chains. `sb_mb` points to the first mbuf in the chain, and `sb_cc` is the total number of data bytes contained within the mbufs. `sb_hiwater` and `sb_lowat` regulate the socket flow control algorithms, `sb_mbcnt` is the total amount of memory allocated to the mbufs in the buffer.

Recall that each mbuf may store from 0 to 2048 bytes of data (if an external cluster is used). `sb_mbmax` is an upper bound on the amount of memory to be allocated as mbufs for each socket buffer. Default limits are specified by each protocol when the `PRU_ATTACH` request is issued by the socket system call. The high-water and low-water marks may be modified by the process as long as the kernel-enforced hard limit of 262,144 bytes per socket buffer (`sb_max`) is not exceeded. The buffering algorithms are described in [Sections 16.7](#) and [16.12](#). [Figure 16.4](#) shows the default settings for the Internet protocols.

## Figure 16.4. Default socket buffer limits for the Internet protocols.

Protocol	so_snd			so_rcv		
	sb_hiwat	sb_lowat	sb_mbmax	sb_hiwat	sb_lowat	sb_mbmax
UDP	$9 \times 1024$	2048 (ignored)	$2 \times \text{sb\_hiwat}$	$40 \times (1024 + 16)$	1	$2 \times \text{sb\_hiwat}$
TCP	$8 \times 1024$	2048	$2 \times \text{sb\_hiwat}$	$8 \times 1024$	1	$2 \times \text{sb\_hiwat}$
raw IP						
ICMP						
IGMP						

Since the source address of each incoming UDP datagram is queued with the data ([Section 23.8](#)), the default UDP value for sb\_hiwat is set to accommodate 40 1K datagrams and their associated sockaddr\_in structures (16 bytes each).

79

sb\_sel is a selinfo structure used to implement the select system call ([Section 16.13](#)).

80

[Figure 16.5](#) lists the possible values for sb\_flags.

## Figure 16.5. sb\_flags values.

sb_flags	Description
<i>SB_LOCK</i>	a process has locked the socket buffer
<i>SB_WANT</i>	a process is waiting to lock the buffer
<i>SB_WAIT</i>	a process is waiting for data (receive) or space (send) in this buffer
<i>SB_SEL</i>	one or more processes are selecting on this buffer
<i>SB_ASYNC</i>	generate asynchronous I/O signal for this buffer
<i>SB_NOINTR</i>	signals do not cancel a lock request
<i>SB_NOTIFY</i>	( <i>SB_WAIT</i>   <i>SB_SEL</i>   <i>SB_ASYNC</i> ) a process is waiting for changes to the buffer and should be notified by wakeup when any changes occur

81-82

`sb_timeo` is measured in clock ticks and limits the time a process blocks during a read or write call. The default value of 0 causes the process to wait indefinitely. `sb_timeo` may be changed or retrieved by the `SO_SNDFTIMEO` and `SO_RCVTIMEO` socket options.

## Socket Macros and Functions

There are many macros and functions that manipulate the send and receive buffers associated with each socket. The macros and functions in [Figure 16.6](#) handle buffer locking and synchronization.

**Figure 16.6. Macros and functions for socket buffer locking and**

## synchronization.

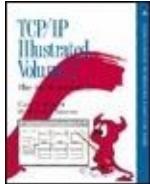
Name	Description
sblock	Acquires a lock for <i>sb</i> . If <i>wf</i> is M_WAITOK, the process sleeps waiting for the lock; otherwise EWOULDBLOCK is returned if the buffer cannot be locked immediately. EINTR or ERESTART is returned if the sleep is interrupted by a signal; 0 is returned otherwise.  <code>int sblock(struct sockbuf *sb, int wf);</code>
sbunlock	Releases the lock on <i>sb</i> . Any other process waiting to lock <i>sb</i> is awakened.  <code>void sbunlock(struct sockbuf *sb);</code>
sbswait	Calls tsleep to wait for protocol activity on <i>sb</i> . Returns result of tsleep.  <code>int sbswait(struct sockbuf *sb);</code>
sowakeup	Notifies socket of protocol activity. Wakes up matching call to sbswait or to tsleep if any processes are selecting on <i>sb</i> .  <code>void sowakeup(struct socket *so, struct sockbuf *sb);</code>
sorwakeup	Wakes up any process waiting for read events on <i>so</i> and sends the SIGIO signal if a process requested asynchronous notification of I/O.  <code>void sorwakeup(struct socket *so);</code>
sowakeup	Wakes up any process waiting for write events on <i>so</i> and sends the SIGIO signal if a process requested asynchronous notification of I/O.  <code>void sowakeup(struct socket *so);</code>

Figure 16.7 includes the macros and functions used to set the resource limits for socket buffers and to append and delete data from the buffers. In the table, *m*, *m0*, *n*, and *control* are all pointers to mbuf chains. *sb* points to the send or receive buffer for a socket.

## Figure 16.7. Macros and functions for socket buffer allocation and manipulation.

Name	Description
sbspace	The number of bytes that may be added to <i>sb</i> before it is considered full: $\min((\text{sb\_hiwat} - \text{sb\_cc}), (\text{sb\_mbmax} - \text{sb\_mbcnt}))$ . <pre>long <b>sbspace</b>(struct sockbuf *sb);</pre>
sballoc	<i>m</i> has been added to <i>sb</i> . Adjust <i>sb_cc</i> and <i>sb_mbcnt</i> in <i>sb</i> accordingly. <pre>void <b>sballoc</b>(struct sockbuf *sb, struct mbuf *m);</pre>
sbfree	<i>m</i> has been removed from <i>sb</i> . Adjust <i>sb_cc</i> and <i>sb_mbcnt</i> in <i>sb</i> accordingly. <pre>int <b>sbfree</b>(struct sockbuf *sb, struct mbuf *m);</pre>
sbappend	Append the mbufs in <i>m</i> to the end of the last record in <i>sb</i> . Call sbcompress. <pre>int <b>sbappend</b>(struct sockbuf *sb, struct mbuf *m);</pre>
sbappendrecord	Append the record in <i>m0</i> after the last record in <i>sb</i> . Call sbcompress. <pre>int <b>sbappendrecord</b>(struct sockbuf *sb, struct mbuf *m0);</pre>
sbappendaddr	Put address from <i>asa</i> in an mbuf. Concatenate address, <i>control</i> , and <i>m0</i> . Append the resulting mbuf chain after the last record in <i>sb</i> . <pre>int <b>sbappendaddr</b>(struct sockbuf *sb, struct sockaddr *asa,                   struct mbuf *m0, struct mbuf *control);</pre>
sbappendcontrol	Concatenate <i>control</i> and <i>m0</i> . Append the resulting mbuf chain after the last record in <i>sb</i> . <pre>int <b>sbappendcontrol</b>(struct sockbuf *sb, struct mbuf *m0,                      struct mbuf *control);</pre>
sbinsertoob	Insert <i>m0</i> before first record in <i>sb</i> without out-of-band data. Call sbcompress. <pre>int <b>sbinsertoob</b>(struct sockbuf *sb, struct mbuf *m0);</pre>
sbcompress	Append <i>m</i> to <i>n</i> squeezing out any unused space. <pre>void <b>sbcompress</b>(struct sockbuf *sb, struct mbuf *m,                  struct mbuf *n);</pre>
sbdrop	Discard <i>len</i> bytes from the front of <i>sb</i> . <pre>void <b>sbdrop</b>(struct sockbuf *sb, int len);</pre>
sbdroprecord	Discard the first record in <i>sb</i> . Move the next record to the front. <pre>void <b>sbdroprecord</b>(struct sockbuf *sb);</pre>
sbrelease	Call sbflush to release all mbufs in <i>sb</i> . Reset <i>sb_hiwat</i> and <i>sb_mbmax</i> values to 0. <pre>void <b>sbrelease</b>(struct sockbuf *sb);</pre>
sbflush	Release all mbufs in <i>sb</i> . <pre>void <b>sbflush</b>(struct sockbuf *sb);</pre>
soreserve	Set high-water and low-water marks. For the send buffer, call sbreserve with <i>sdcc</i> . For the receive buffer, call sbreserve with <i>rcvc</i> . Initialize <i>sb_lowat</i> in both buffers to default values, Figure 16.4. ENOBUFFS is returned if any limits are exceeded. <pre>int <b>soreserve</b>(struct socket *so, int sdcc, int rcvc);</pre>
sbreserve	Set high-water mark for <i>sb</i> to <i>cc</i> . Also drop low-water mark to <i>cc</i> . No memory is allocated by this function. <pre>int <b>sbreserve</b>(struct sockbuf *sb, int cc);</pre>

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

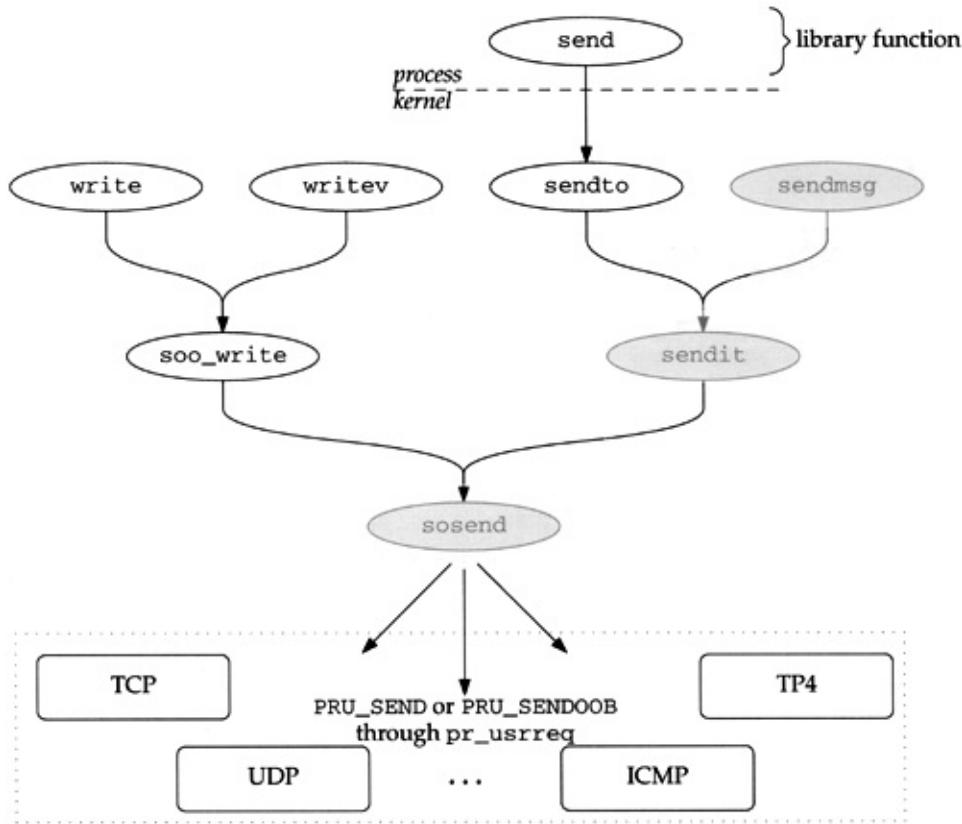
### 16.4 write, writev, sendto, and sendmsg System Calls

These four system calls, which we refer to collectively as the *write system calls*, send data on a network connection. The first three system calls are simpler interfaces to the most general request, sendmsg.

All the write system calls, directly or indirectly, call sosend, which does the work of copying data from the process to the kernel and passing data to the protocol associated with the socket. [Figure 16.8](#) summarizes the flow of control.

**Figure 16.8. All socket output is handled**

## by sosend.



In the following sections, we discuss the functions shaded in [Figure 16.8](#). The other four system calls and `soo_write` are left for readers to investigate on their own.

[Figure 16.9](#) shows the features of these four system calls and a related library function (`send`).

**Figure 16.9. Write system calls.**

Function	Type of descriptor	Number of buffers	Specify destination address?	Flags?	Control information?
write	any	1			
writev	any	[1..UIO_MAXIOV]			
send	socket only	1		:	
sendto	socket only	1		:	
sendmsg	socket only	[1..UIO_MAXIOV]	:	:	.

In Net/3, send is implemented as a library function that calls sendto. For binary compatibility with previously compiled programs, the kernel maps the old send system call to the function osend, which is not discussed in this text.

From the second column in [Figure 16.9](#) we see that the write and writev system calls are valid with any descriptor, but the remaining system calls are valid only with socket descriptors.

The third column shows that writev and sendmsg accept data from multiple buffers. Writing from multiple buffers is called *gathering*. The analogous read operation is called *scattering*. In a gather operation the kernel accepts, in order, data from each buffer specified in an array of iovec structures. The array can have a maximum of UIO\_MAXIOV elements. The structure is shown in [Figure 16.10](#).

## Figure 16.10. iovec structure.

```
41 struct iovec {  
42     char *iov_base;           /* Base address */  
43     size_t iov_len;          /* Length */  
44 };
```

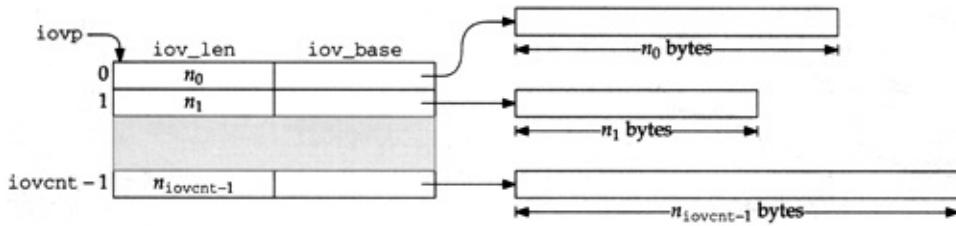
41-44

iov\_base points to the start of a buffer of iov\_len bytes.

Without this type of interface, a process would have to copy buffers into a single larger buffer or make multiple write system calls to send data from multiple buffers. Both alternatives are less efficient than passing an array of iovec structures to the kernel in a single call. With datagram protocols, the result of one writev is one datagram, which cannot be emulated with multiple writes.

Figure 16.11 illustrates the structures as they are used by writev, where iovp points to the first element of the array and iovcnt is the size of the array.

## Figure 16.11. iovec arguments to writev.



Datagram protocols require a destination address to be associated with each write call. Since `write`, `writev`, and `send` do not accept an explicit destination, they may be called only after a destination has been associated with a connectionless socket by calling `connect`. A destination must be provided with `sendto` or `sendmsg`, or `connect` must have been previously called.

The fifth column in [Figure 16.9](#) shows that the `sendxxx` system calls accept optional control flags, which are described in [Figure 16.12](#).

## Figure 16.12. `sendxxx` system calls: flags values.

flags	Description	Reference
<code>MSG_DONTROUTE</code>	bypass routing tables for this message	<a href="#">Figure 16.23</a>
<code>MSG_DONTWAIT</code>	do not wait for resources during this message	<a href="#">Figure 16.22</a>
<code>MSG_EOR</code>	data marks the end of a logical record	<a href="#">Figure 16.25</a>
<code>MSG_OOB</code>	send as out-of-band data	<a href="#">Figure 16.26</a>

As indicated in the last column of [Figure 16.9](#), only the sendmsg system call supports control information. The control information and several other arguments to sendmsg are specified within a msghdr structure ([Figure 16.13](#)) instead of being passed separately.

**Figure 16.13. msghdr structure.**

```
228 struct msghdr {  
229     caddr_t msg_name;           /* optional address */  
230     u_int    msg_namelen;        /* size of address */  
231     struct iovec *msg iov;      /* scatter/gather array */  
232     u_int    msg iovlen;         /* # elements in msg iov */  
233     caddr_t msg_control;        /* ancillary data, see below */  
234     u_int    msg_controllen;     /* ancillary data buffer len */  
235     int     msg_flags;          /* Figure 16.33 */  
236 };
```

socket.h

socket.h

msg\_name should be declared as a pointer to a sockaddr structure, since it contains a network address.

228-236

The msghdr structure contains a destination address (msg\_name and msg\_namelen), a scatter/gather array (msg iov and msg iovlen), control information (msg\_control and msg\_controllen), and receive flags

(msg\_flags). The control information is formatted as a cmsghdr structure shown in Figure 16.14.

**Figure 16.14. cmsghdr structure.**

```
-----socket.h
251 struct cmsghdr {
252     u_int    cmsg_len;           /* data byte count, including hdr */
253     int      cmsg_level;        /* originating protocol */
254     int      cmsg_type;         /* protocol-specific type */
255 /* followed by u_char cmsg_data[]; */
256 };
-----socket.h
```

251-256

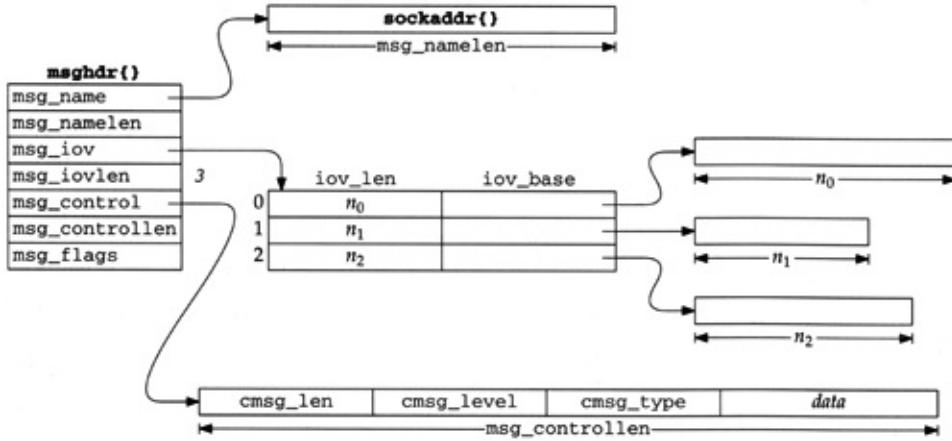
The control information is not interpreted by the socket layer, but the messages are typed (cmsg\_type) and they have an explicit length (cmsg\_len). Multiple control messages may appear in the control information mbuf.

## Example

Figure 16.15 shows how a fully specified msghdr structure might look during a call to sendmsg.

**Figure 16.15. msghdr structure for**

# sendmsg system call.

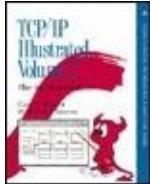


Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.5 sendmsg System Call

Only the sendmsg system call provides access to all the features of the sockets API associated with output. The sendmsg and sendit functions prepare the data structures needed by sosend, which passes the message to the appropriate protocol. For SOCK\_DGRAM protocols, a message is a datagram. For SOCK\_STREAM protocols, a message is a sequence of bytes. For SOCK\_SEQPACKET protocols, a message could be an entire record (implicit record boundaries) or part of a larger record (explicit record boundaries). A message is always an entire record (implicit record boundaries) for SOCK\_RDM protocols.

Even though the general sosend code handles SOCK\_SEQPACKET and SOCK\_RDM protocols, there are no such protocols in the Internet domain.

Figure 16.16 shows the sendmsg code.

## Figure 16.16. sendmsg system call.

```
307 struct sendmsg_args {
308     int      s;
309     caddr_t  msg;
310     int      flags;
311 };
312 sendmsg(p, uap, retval)
313 struct proc *p;
314 struct sendmsg_args *uap;
315 int      *retval;
316 {
317     struct msghdr msg;
318     struct iovec aiov[UIO_SMALLIOV], *iov;
319     int      error;
320     if (error = copyin(uap->msg, (caddr_t) &msg, sizeof(msg)))
321         return (error);
322     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
323         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
324             return (EMSGSIZE);
325         MALLOC(iov, struct iovec *,
326                sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
327                M_WAITOK);
328     } else
329         iov = aiov;
330     if (msg.msg_iovlen &&
331         (error = copyin((caddr_t) msg.msg_iov, (caddr_t) iov,
332                         (unsigned) (msg.msg_iovlen * sizeof(struct iovec))))) {
333         goto done;
334     msg.msg_iov = iov;
335     error = sendit(p, uap->s, &msg, uap->flags, retval);
336     done:
337     if (iov != aiov)
338         FREE(iov, M_IOV);
339     return (error);
340 }
```

There are three arguments to sendmsg: the socket descriptor; a pointer to a msghdr structure; and several control flags. The copyin function copies the msghdr structure from user space to the kernel.

## Copy iov array

322-334

An iovec array with eight entries (UIO\_SMALLIOV) is allocated automatically on the stack. If this is not large enough, sendmsg calls MALLOC to allocate a larger array. If the process specifies an array with more than 1024 (UIO\_MAXIOV) entries, EMSGSIZE is returned. copyin places a copy of the iovec array from user space into either the array on the stack or the larger, dynamically allocated, array.

This technique avoids the relatively expensive call to malloc in the most common case of eight or fewer entries.

## sendit and cleanup

## 335-340

When sendit returns, the data has been delivered to the appropriate protocol or an error has occurred. sendmsg releases the iovec array (if it was dynamically allocated) and returns sendit's result.

---

[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)

## Chapter 16. Socket I/O

### 16.6 sendit Function

sendit is the common function called by sendto initializes a uio structure and copies control and from the process into the kernel. Before discussing explain the uiomove function and the uio struct

### uiomove Function

The prototype for this function is:

```
int uiomove(caddr_t cp, int n, s
```

The uiomove function moves  $n$  bytes between a referenced by  $cp$  and the multiple buffers specified in  $uio$ . [Figure 16.17](#) shows the definition of the controls and records the actions of the uiomove

## Figure 16.17. uio structure

```
45 enum uio_rw {                                     uio.h
46     UIO_READ, UIO_WRITE
47 };
48 enum uio_seg {                                /* Segment flag values */
49     UIO_USERSPACE,          /* from user data space */
50     UIO_SYSSPACE,          /* from system space */
51     UIO_USERISPACE         /* from user instruction space */
52 };
53 struct uio {
54     struct iovec *uio_iov;    /* an array of iovec structures */
55     int      uio_iovcnt;     /* size of iovec array */
56     off_t   uio_offset;      /* starting position of transfer */
57     int      uio_resid;      /* remaining bytes to transfer */
58     enum uio_seg uio_segflg; /* location of buffers */
59     enum uio_rw uio_rw;       /* direction of transfer */
60     struct proc *uio_procp;  /* the associated process */
61 };
```

45-61

In the uio structure, uio iov points to an array uio offset counts the number of bytes transferred uio resid counts the number of bytes remaining. Each time uiomove is called, uio offset increases or decreases by  $n$ . uiomove adjusts the base pointer lengths in the uio iov array to exclude any bytes transferred each time it is called. Finally, uio iov points to each entry in the array as each buffer is transferred. uio segflg indicates the location of the buffers specified by the uio iov array and uio rw indicates the direction. The buffers may be located in the user data space, kernel data space, or system space. [Figure 16.18](#) summarizes the arguments of uiomove. The descriptions use the argument types of the uiomove prototype.

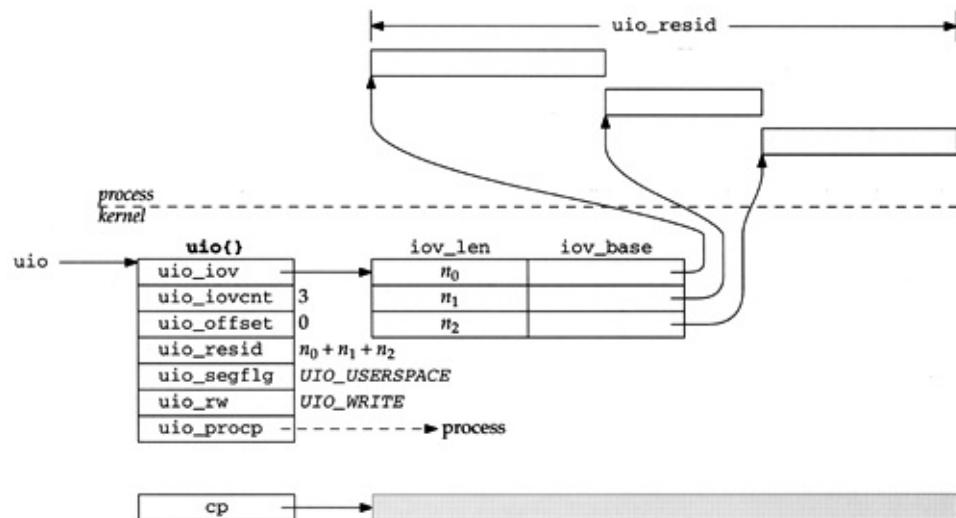
**Figure 16.18. uiomove operation**

uio_segflg	uio_rw	Description
<i>UIO_USERSPACE</i>	<i>UIO_READ</i>	scatter $n$ bytes from a kernel buffer $cp$ to process buffers
<i>UIO_USERISPACE</i>		
<i>UIO_USERSPACE</i>	<i>UIO_WRITE</i>	gather $n$ bytes from process buffers into the kernel buffer $cp$
<i>UIO_USERISPACE</i>		
<i>UIO_SYSSPACE</i>	<i>UIO_READ</i>	scatter $n$ bytes from the kernel buffer $cp$ to multiple kernel buffers
	<i>UIO_WRITE</i>	gather $n$ bytes from multiple kernel buffers into the kernel buffer $cp$

## Example

Figure 16.19 shows a uio structure before uiomove.

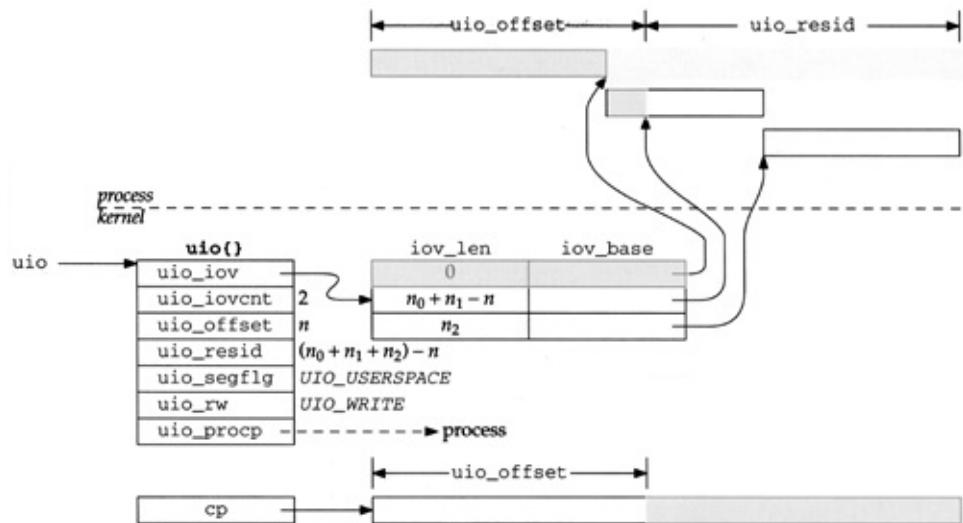
**Figure 16.19. uiomove: before**



`uio iov` points to the first entry in the `iovec` array.  
`iov_base` pointers point to the start of their respective buffers.

address space of the process. `uio_offset` is 0, a sum of size of the three buffers. `cp` points to a kernel, typically the data area of an mbuf. [Figure 16.19](#) shows the same data structures after

**Figure 16.20. uiomove: after**



`uiomove(cp, n, uio);`

is executed where  $n$  includes all the bytes from the first buffer ( $n_0$ ) and only some of the bytes from the second buffer ( $n_1$ ).

After `uiomove`, the first buffer has a length of 0 and has been advanced to the end of the buffer. `uio_offset` now points to the second entry in the `iovec` array. The pointer `cp` has been advanced and the length decreased to reflect the amount of data moved.

some of the bytes in the buffer. `uio_offset` has increased by  $n$  and `uio_resid` has been decreased by  $n$ . The data has been moved into the kernel's buffer. The operation was `UIO_WRITE`.

## sendit Code

We can now discuss the `sendit` code shown in Figure 16.21.

**Figure 16.21. sendit function**

---

```

341 sendit(p, s, mp, flags, retsize)           — uipc_syscalls.c
342 struct proc *p;
343 int     s;
344 struct msghdr *mp;
345 int     flags, *retsize;
346 {
347     struct file *fp;
348     struct uio auio;
349     struct iovec *iov;
350     int      i;
351     struct mbuf *to, *control;
352     int      len, error;
353     if (error = getsock(p->p_fd, s, &fp))
354         return (error);
355     auio.uio_iov = mp->msg_iov;
356     auio.uio_iovlen = mp->msg_iovlen;
357     auio.uio_segflg = UIO_USERSPACE;
358     auio.uio_rw = UIO_WRITE;
359     auio.uio_procp = p;
360     auio.uio_offset = 0;          /* XXX */
361     auio.uio_resid = 0;
362     iov = mp->msg_iov;
363     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
364         if (iov->iov_len < 0)
365             return (EINVAL);
366         if ((auio.uio_resid += iov->iov_len) < 0)
367             return (EINVAL);
368     }
369     if (mp->msg_name) {
370         if (error = sockargs(&to, mp->msg_name, mp->msg_namelen,
371                             MT_SONAME))
372             return (error);
373     } else
374         to = 0;
375     if (mp->msg_control) {
376         if (mp->msg_controllen < sizeof(struct cmsghdr))
377             {
378                 error = EINVAL;
379                 goto bad;
380             }
381         if (error = sockargs(&control, mp->msg_control,
382                             mp->msg_controllen, MT_CONTROL))
383             goto bad;
384     } else
385         control = 0;
386     len = auio.uio_resid;
387     if (error = sosend((struct socket *) fp->f_data, to, &auio,
388                         (struct mbuf *) 0, control, flags)) {
389         if (auio.uio_resid != len && (error == ERESTART ||
390                                         error == EINTR || error == EWOULDBLOCK))
391             error = 0;
392         if (error == EPIPE)
393             psignal(p, SIGPIPE);
394     }
395     if (error == 0)
396         *retsize = len - auio.uio_resid;
397     bad:
398     if (to)
399         m_freem(to);
400     return (error);
401 }

```

---

— uipc\_syscalls.c

## Initialize auio

341-368

sendit calls getsock to get the file structure associated with descriptor s and initializes the uio structure to contain buffers specified by the process into mbufs in theiov array. The length of the transfer is calculated by the for loop as the sum of the buffer lengths and saved in uio\_resid. The first if within the loop checks that the buffer length is nonnegative. The second if checks that uio\_resid does not overflow, since uio\_resid is initialized to zero. The third if checks that iov\_len is guaranteed to be nonnegative.

## **Copy address and control information from the**

369-385

`sockargs` makes copies of the destination address information into mbufs if they are provided by the user.

# Send data and cleanup

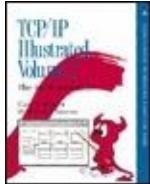
386-401

uio\_resid is saved in len so that the number of bytes to be calculated if sosend does not accept all the bytes. The destination address, uio structure, control information, and flags are all passed to sosend. When sosend returns, serial follows:

- If sosend transfers some data and is interrupted by a blocking condition, the error is discarded and no error is reported.
- If sosend returns EPIPE, the SIGPIPE signal is delivered to the process. error is not set to 0, so if a process calls the signal handler returns, or if the process exits, the write call returns EPIPE.
- If no error occurred (or it was discarded), the number of bytes transferred is calculated and saved in \*retsize. If error returns 0, syscall ([Section 15.4](#)) returns \*retsize instead of returning the error code.
- If any other error occurs, the error code is returned to the process.

Before returning, sendit releases the mbuf containing the destination address. sosend is responsible for releasing the mbuf.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.7 sosend Function

sosend is one of the most complicated functions in the socket layer. Recall from [Figure 16.8](#) that all five write calls eventually call sosend. It is sosend's responsibility to pass the data and control information to the pr\_usrreq function of the protocol associated with the socket according to the semantics supported by the protocol and the buffer limits specified by the socket. sosend never places data in the send buffer; it is the protocol's responsibility to store and remove the data.

The interpretation of the send buffer's sb\_hiwat and sb\_lowat values by sosend

depends on whether the associated protocol implements reliable or unreliable data transfer semantics.

## Reliable Protocol Buffering

For reliable protocols, the send buffer holds both data that has not yet been transmitted and data that has been sent, but has not been acknowledged. `sb_cc` is the number of bytes of data that reside in the send buffer, and  $0 \leq sb\_cc \leq sb\_hiwat$ .

`sb_cc` may temporarily exceed `sb_hiwat` when out-of-band data is sent.

It is `sosend`'s responsibility to ensure that there is enough space in the send buffer before passing any data to the protocol layer through the `pr_usrreq` function. The protocol layer adds the data to the send buffer. `sosend` transfers data to the protocol in one of two ways:

- If `PR_ATOMIC` is set, `sosend` must preserve the message boundaries between the process and the protocol

layer. In this case, `sosend` waits for enough space to become available to hold the entire message. When the space is available, an mbuf chain containing the entire message is constructed and passed to the protocol in a single call through the `pr_usrreq` function. RDP and SPP are examples of this type of protocol.

- If `PR_ATOMIC` is not set, `sosend` passes the message to the protocol one mbuf at a time and may pass a partial mbuf to avoid exceeding the high-water mark. This method is used with `SOCK_STREAM` protocols such as TCP and `SOCK_SEQPACKET` protocols such as TP4. With TP4, record boundaries are indicated explicitly with the `MSG_EOR` flag ([Figure 16.12](#)), so it is not necessary for the message boundaries to be preserved by `sosend`.

TCP applications have no control over the size of outgoing TCP segments. For example, a message of 4096 bytes sent on a TCP socket will be split by the socket layer into two mbufs with external

clusters, containing 2048 bytes each, assuming there is enough space in the send buffer for 4096 bytes. Later, during protocol processing, TCP will segment the data according to the maximum segment size for the connection, which is normally less than 2048.

When a message is too large to fit in the available buffer space and the protocol allows messages to be split, `sosend` still does not pass data to the protocol until the free space in the buffer rises above `sb_lowat`. For TCP, `sb_lowat` defaults to 2048 ([Figure 16.4](#)), so this rule prevents the socket layer from bothering TCP with small chunks of data when the send buffer is nearly full.

## Unreliable Protocol Buffering

With unreliable protocols (e.g., UDP), no data is ever stored in the send buffer and no acknowledgment is ever expected. Each message is passed immediately to the protocol where it is queued for transmission on the appropriate network

device. In this case, `sb_cc` is always 0, and `sb_hiwat` specifies the maximum size of each write and indirectly the maximum size of a datagram.

[Figure 16.4](#) shows that `sb_hiwat` defaults to 9216 ( $9 \times 1024$ ) for UDP. Unless the process changes `sb_hiwat` with the `SO_SNDBUF` socket option, an attempt to write a datagram larger than 9216 bytes returns with an error. Even then, other limitations of the protocol implementation may prevent a process from sending large datagrams. Section 11.10 of Volume 1 discusses these defaults and limits in other TCP/IP implementations.

9216 is large enough for a NFS write, which often defaults to 8192 bytes of data plus protocol headers.

## sosend Code

[Figure 16.22](#) shows an overview of the `sosend` function. We discuss the four shaded sections separately.

## Figure 16.22. sosend function: overview.

```
271 sosend(so, addr, uio, top, control, flags) ----- uipc_socket.c
272 struct socket *so;
273 struct mbuf *addr;
274 struct uio *uio;
275 struct mbuf *top;
276 struct mbuf *control;
277 int     flags;
278 {

    /* initialization (Figure 16.23) */

305     restart:
306     if (error = sblock(&so->so_snd, SBLOCKWAIT(flags)))
307         goto out;
308     do {                                /* main loop, until resid == 0 */

        /* wait for space in send buffer (Figure 16.24) */

342     do {
343         if (uio == NULL) {
344             /*
            * Data is prepackaged in "top".
            */
345             resid = 0;
346             if (flags & MSG_EOR)
347                 top->m_flags |= M_EOR;
348         } else
349             do {

                /* fill a single mbuf or an mbuf chain (Figure 16.25) */

396         } while (space > 0 && atomic);

            /* pass mbuf chain to protocol (Figure 16.26) */

412     } while (resid && space > 0);
413     } while (resid);

414     release:
415     sbunlock(&so->so_snd);
416     out:
417     if (top)
418         m_free(top);
419     if (control)
420         m_free(control);
421     return (error);
422 }
```

271-278

The arguments to sosend are: so, a pointer to the relevant socket; addr, a

pointer to a destination address; uio, a pointer to a uio structure describing the I/O buffers in user space; top, an mbuf chain that holds data to be sent; control, an mbuf that holds control information to be sent; and flags, which contains options for this write call.

Normally, a process provides data to the socket layer through the uio mechanism and top is null. When the kernel itself is using the socket layer (such as with NFS), the data is passed to sosend as an mbuf chain pointed to by top, and uio is null.

279-304

The initialization code is described separately.

## Lock send buffer

305-308

sosend's main processing loop starts at restart, where it obtains a lock on the send buffer with sblock before proceeding. The lock ensures orderly access to the socket

buffer by multiple processes.

If `MSG_DONTWAIT` is set in flags, then `SBLOCKWAIT` returns `M_NOWAIT`, which tells `sblock` to return `EWOULDBLOCK` if the lock is not available immediately.

`MSG_DONTWAIT` is used only by NFS in Net/3.

The main loop continues until `sosend` transfers all the data to the protocol (i.e., `resid == 0`).

## Check for space

309-341

Before any data is passed to the protocol, various error conditions are checked and `sosend` implements the flow control and resource control algorithms described earlier. If `sosend` blocks waiting for more space to appear in the output buffer, it jumps back to restart before continuing.

## Use data from top

342-350

Once space becomes available and sosend has obtained a lock on the send buffer, the data is prepared for delivery to the protocol layer. If uio is null (i.e., the data is in the mbuf chain pointed to by top), sosend checks MSG\_EOR and sets M\_EOR in the chain to mark the end of a logical record. The mbuf chain is ready for the protocol layer.

## Copy data from process

351-396

When uio is not null, sosend must transfer the data from the process. When PR\_ATOMIC is set (e.g., UDP), this loop continues until all the data has been stored in a single mbuf chain. A break, which is not shown in [Figure 16.22](#), causes the loop to terminate when all the data has been copied from the process, and sosend passes the entire chain to the protocol.

When PR\_ATOMIC is not set (e.g., TCP), this loop is executed only once, filling a single mbuf with data from uio. In this case, the mbufs are passed one at a time to the protocol.

## Pass data to the protocol

397-413

For PR\_ATOMIC protocols, after the mbuf chain is passed to the protocol, resid is always 0 and control falls through the two loops to release. When PR\_ATOMIC is not set, sosend continues filling individuals mbufs while there is more data to send and while there is still space in the buffer. If the buffer fills and there is still data to send, sosend loops back and waits for more space before filling the next mbuf. If all the data is sent, both loops terminate.

## Cleanup

414-422

After all the data has been passed to the

protocol, the socket buffer is unlocked, any remaining mbufs are discarded, and sosend returns.

The detailed description of sosend is shown in four parts:

- initialization (Figure 16.23),

### Figure 16.23. sosend function: initialization.

```
279     struct proc *p = curproc; /* XXX */
280     struct mbuf **mp;
281     struct mbuf *m;
282     long    space, len, resid;
283     int      clen = 0, error, s, dontroute, mlen;
284     int      atomic = sosendallatonce(so) || top;
285
286     if (uio)
287         .resid = uio->uio_resid;
288     else
289         resid = top->m_pkthdr.len;
290
291     /*
292      * In theory resid should be unsigned.
293      * However, space must be signed, as it might be less than 0
294      * if we over-committed, and we must use a signed comparison
295      * of space and resid. On the other hand, a negative resid
296      * causes us to loop sending 0-length segments to the protocol.
297      */
298     if (resid < 0)
299         return (EINVAL);
300     dontroute =
301         (flags & MSG_DONTROUTE) && (so->so_options & SO_DONTROUTE) == 0 &&
302         (so->so_proto->pr_flags & PR_ATOMIC);
303     p->p_stats->p_ru.ru_msgsnd++;
304     if (control)
305         clen = control->m_len;
306 #define snderr(errno) { error = errno; splx(s); goto release; }
```

- error and resource checking (Figure 16.24),

## Figure 16.24. sosend function: error and resource checking.

```
309     s = splnet();                                     uipc_socket.c
310     if (so->so_state & SS_CANTSENDMORE)
311         snderr(EPIPE);
312     if (so->so_error)
313         snderr(so->so_error);
314     if ((so->so_state & SS_ISCONNECTED) == 0) {
315         if (so->so_proto->pr_flags & PR_CONNREQUIRED) {
316             if ((so->so_state & SS_ISCONFIRMING) == 0 &&
317                 !(resid == 0 && clen != 0))
318                 snderr(ENOTCONN);
319             } else if (addr == 0)
320                 snderr(EDESTADDRREQ);
321         }
322     space = sbspace(&so->so_snd);
323     if (flags & MSG_OOB)
324         space += 1024;
325     if (atomic && resid > so->so_snd.sb_hiwat ||
326         clen > so->so_snd.sb_hiwat)
327         snderr(EMSGSIZE);
328     if (space < resid + clen && uio &&
329         (atomic || space < so->so_snd.sb_lowat || space < clen)) {
330         if (so->so_state & SS_NBIO)
331             snderr(EWOULDBLOCK);
332         sbunlock(&so->so_snd);
333         error = sbwait(&so->so_snd);
334         splx(s);
335         if (error)
336             goto out;
337         goto restart;
338     }
339     splx(s);
340     mp = &top;
341     space -= clen;
```

- data transfer (Figure 16.25), and

## Figure 16.25. sosend function: data transfer.

```

351             do {
352                 if (top == 0) {
353                     MGETHDR(m, M_WAIT, MT_DATA);
354                     mlen = MHLEN;
355                     m->m_pkthdr.len = 0;
356                     m->m_pkthdr.rcvif = (struct ifnet *) 0;
357                 } else {
358                     MGET(m, M_WAIT, MT_DATA);
359                     mlen = MLEN;
360                 }
361
362                 if (resid >= MINCLSIZE && space >= MCLBYTES) {
363                     MCLGET(m, M_WAIT);
364                     if ((m->m_flags & M_EXT) == 0)
365                         goto nopages;
366                     mlen = MCLBYTES;
367                     if (atomic && top == 0) {
368                         len = min(MCLBYTES - max_hdr, resid);
369                         m->m_data += max_hdr;
370                     } else
371                         len = min(MCLBYTES, resid);
372                     space -= MCLBYTES;
373                 } else {
374                     nopages:
375                         len = min(min(mlen, resid), space);
376                         space -= len;
377                         /*
378                             * For datagram protocols, leave room
379                             * for protocol headers in first mbuf.
380                             */
381                         if (atomic && top == 0 && len < mlen)
382                             MH_ALIGN(m, len);
383
384                         error = uiomove(mtod(m, caddr_t), (int) len, uio);
385                         resid = uio->uio_resid;
386                         m->m_len = len;
387                         *mp = m;
388                         top->m_pkthdr.len += len;
389                         if (error)
390                             goto release;
391                         mp = &m->m_next;
392                         if (resid <= 0) {
393                             if (flags & MSG_EOR)
394                                 top->m_flags |= M_EOR;
395                             break;
396                         }
397                     } while (space > 0 && atomic);

```

*uipc\_socket.c*

- protocol dispatch (Figure 16.26).

## Figure 16.26. sosend function: protocol dispatch.

```

397         if (dontroute)
398             so->so_options |= SO_DONTROUTE;
399             s = splnet(); /* XXX */
400             error = (*so->so_proto->pr_usrreq) (so,
401                                         (flags & MSG_OOB) ? PRU_SENDOOB : PRU_SEND,
402                                         top, addr, control);
403             splx(s);
404             if (dontroute)
405                 so->so_options &= ~SO_DONTROUTE;
406             clen = 0;
407             control = 0;
408             top = 0;
409             mp = &top;
410             if (error)
411                 goto release;
412             } while (resid && space > 0);
413         } while (resid);

```

*uipc\_socket.c*

The first part of sosend shown in [Figure 16.23](#) initializes various variables.

## Compute transfer size and semantics

279-284

atomic is set if sosendallatonce is true (any protocol for which PR\_ATOMIC is set) or the data has been passed to sosend as an mbuf chain in top. This flag controls whether data is passed to the protocol as a single mbuf chain or in separate mbufs.

285-297

resid is the number of bytes in the iovec buffers or the number of bytes in the top mbuf chain. [Exercise 16.1](#) discusses why

resid might be negative.

## If requested, disable routing

298-303

dontroute is set when the routing tables should be bypassed for *this* message only. clen is the number of bytes in the optional control mbuf.

304

The macro snderr posts the error code, reenables protocol processing, and jumps to the cleanup code at out. This macro simplifies the error handling within the function.

[Figure 16.24](#) shows the part of sosend that checks for error conditions and waits for space to appear in the send buffer.

309

Protocol processing is suspended to prevent the buffer from changing while it is being examined. Before each transfer,

sosend checks several conditions:

- **310-311**

If output from the socket is prohibited (e.g., the write-half of a TCP connection has been closed), EPIPE is returned.

- **312-313**

If the socket is in an error state (e.g., an ICMP port unreachable may have been generated by a previous datagram), so\_error is returned. sendit discards the error if some data has been sent before the error occurs (Figure 16.21, line 389).

- **314-318**

If the protocol requires connections and a connection has not been established or a connection attempt has not been started, ENOTCONN is returned. sosend permits a write consisting of control information and no data even when a connection has not been established.

The Internet protocols do not use

this feature, but it is used by TP4 to send data with a connection request, to confirm a connection request, and to send data with a disconnect request.

- 319-321

If a destination address is not specified for a connectionless protocol (e.g., the process calls send without establishing a destination with connect), EDESTADDREQ is returned.

## Compute available space

### 322-324

sbspace computes the amount of free space remaining in the send buffer. This is an administrative limit based on the buffer's high-water mark, but is also limited by sb\_mbmax to prevent many small messages from consuming too many mbufs ([Figure 16.6](#)). sosend gives out-of-band data some priority by relaxing the limits on the buffer size by 1024 bytes.

## Enforce message size limit

325-327

If atomic is set and the message is larger than the high-water mark, EMSGSIZE is returned; the message is too large to be accepted by the protocol even if the buffer were empty. If the control information is larger than the high-water mark, EMSGSIZE is also returned. This is the test that limits the size of a datagram or record.

## Wait for more space?

328-329

If there is not enough space in the send buffer, the data is from a process (versus from the kernel in top), and one of the following conditions is true, then sosend must wait for additional space before continuing:

- the message must be passed to protocol in a single request (atomic is

set), or

- the message may be split, but the free space has dropped below the low-water mark, or
- the message may be split, but the control information does not fit in the available space.

When the data is passed to sosend in top (i.e., when uio is null), the data is already located in mbufs. Therefore sosend ignores the high- and low-water marks since no additional mbuf allocations are required to pass the data to the protocol.

If the send buffer low-water mark is not used in this test, an interesting interaction occurs between the socket layer and the transport layer that leads to performance degradation. [[Crowcroft et al. 1992](#)] provides details on this scenario.

## Wait for space

330-338

If sosend must wait for space and the socket is nonblocking, EWOULDBLOCK is returned. Otherwise, the buffer lock is released and sosend waits with sbwait until the status of the buffer changes. When sbwait returns, sosend reenables protocol processing and jumps back to restart to obtain a lock on the buffer and to check the error and space conditions again before continuing.

By default, sbwait blocks until data can be sent. By changing sb\_timeo in the buffer through the SO\_SNDTIMEO socket option, the process selects an upper bound for the wait time. If the timer expires, sbwait returns EWOULDBLOCK. Recall from [Figure 16.21](#) that this error is discarded by sendit if some data has already been transferred to the protocol. This timer does not limit the length of the entire call, just the inactivity time between filling mbufs.

339-341

At this point, sosend has determined that some data may be passed to the protocol. splx enables interrupts since they should

not be blocked during the relatively long time it takes to copy data from the process to the kernel. mp holds a pointer used to construct the mbuf chain. The size of the control information (clen) is subtracted from the space available before sosend transfers any data from the process.

[Figure 16.25](#) shows the section of sosend that moves data from the process to one or more mbufs in the kernel.

## **Allocate packet header or standard mbuf**

351-360

When atomic is set, this code allocates a packet header during the first iteration of the loop and standard mbufs afterwards. When atomic is not set, this code always allocates a packet header since top is always cleared before entering the loop.

## **If possible, use a cluster**

361-371

If the message is large enough to make a cluster allocation worthwhile and space is greater than or equal to MCLBYTES, a cluster is attached to the mbuf by MCLGET. When space is less than MCLBYTES, the extra 2048 bytes will break the allocation limit for the buffer since the entire cluster is allocated even if resid is less than MCLBYTES.

If MCLGET fails, sosend jumps to nopages and uses a standard mbuf instead of an external cluster.

The test against MINCLSIZE should use `>`, not `>=`, since a write of 208 (MINCLSIZE) bytes fits within two mbufs.

When atomic is set (e.g., UDP), the mbuf chain represents a datagram or record and max\_hdr bytes are reserved at the front of the *first* cluster for protocol headers. Subsequent clusters are part of the same chain and do not need room for the headers.

If atomic is not set (e.g., TCP), no space is

reserved since sosend does not know how the protocol will segment the outgoing data.

Notice that space is decremented by the size of the cluster (2048 bytes) and not by len, which is the number of data bytes to be placed in the cluster ([Exercise 16.2](#)).

## Prepare the mbuf

372-382

If a cluster was not used, the number of bytes stored in the mbuf is limited by the smaller of: (1) the space in the mbuf, (2) the number of bytes in the message, or (3) the space in the buffer.

When atomic is set, MH\_ALIGN locates the data at the end of the buffer for the first buffer in the chain. MH\_ALIGN is skipped if the data completely fills the mbuf. This may or may not leave enough room for protocol headers, depending on how much data is placed in the mbuf. When atomic is not set, no space is set aside for the

headers.

## Get data from the process

383-395

uiomove copies len bytes of data from the process to the mbuf. After the transfer, the mbuf length is updated, the previous mbuf is linked to the new mbuf (or top points to the first mbuf), and the length of the mbuf chain is updated. If an error occurred during the transfer, sosend jumps to release.

When the last byte is transferred from the process, M\_EOR is set in the packet if the process set MSG\_EOR, and sosend breaks out of this loop.

MSG\_EOR applies only to protocols with explicit record boundaries such as TP4, from the OSI protocol suite. TCP does not support logical records and ignores the MSG\_EOR flag.

## Fill another buffer?

396

If atomic is set, sosend loops back and begins filling another mbuf.

The test for space > 0 appears to be extraneous. space is irrelevant when atomic is not set since the mbufs are passed to the protocol one at a time. When atomic is set, this loop is entered only when there is enough space for the entire message. See also [Exercise 16.2](#).

The last section of sosend, shown in [Figure 16.26](#), passes the data and control mbufs to the protocol associated with the socket.

397-405

The socket's SO\_DONTROUTE option is toggled if necessary before and after passing the data to the protocol layer to bypass the routing tables on this message. This is the only option that can be enabled for a single message and, as described with [Figure 16.23](#), it is controlled by the MSG\_DONTROUTE flag during a write.

pr\_usrreq is bracketed with splnet and splx

to block interrupts while the protocol is processing the message. This is a paranoid assumption since some protocols (such as UDP) may be able to do output processing without blocking interrupts, but this information is not available at the socket layer.

If the process tagged this message as out-of-band data, sosend issues the PRU\_SENDOOB request; otherwise it issues the PRU\_SEND request. Address and control mbufs are also passed to the protocol at this time.

#### 406-413

clen, control, top, and mp are reset, since control information is passed to the protocol only once and a new mbuf chain is constructed for the next part of the message. resid is nonzero only when atomic is not set (e.g., TCP). In that case, if space remains in the buffer, sosend loops back to fill another mbuf. If there is no more space, sosend loops back to wait for more space ([Figure 16.24](#)).

We'll see in [Chapter 23](#) that unreliable protocols, such as UDP, immediately queue the data for transmission on the network. [Chapter 26](#) describes how reliable protocols, such as TCP, add the data to the socket's send buffer where it remains until it is sent to, and acknowledged by, the destination.

## sosend Summary

sosend is a complex function. It is 142 lines long, contains three nested loops, one loop implemented with goto, two code paths based on whether PR\_ATOMIC is set or not, and two concurrency locks. As with much software, some of the complexity has accumulated over the years. NFS added the MSG\_DONTWAIT semantics and the possibility of receiving data from an mbuf chain instead of the buffers in a process. The SS\_ISCONFIRMING state and MSG\_EOR flag were introduced to handle the connection and record semantics of the OSI protocols.

A cleaner approach would be to implement

a separate `sosend` function for each type of protocol and dispatch through a `pr_send` pointer in the `protosw` entry. This idea is suggested and implemented for UDP in [Partridge and Pink 1993].

## Performance Considerations

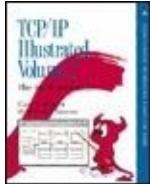
As described in Figure 16.25, `sosend`, when possible, passes message in mbuf-sized chunks to the protocol layer. While this results in more calls to the protocol than building and passing an entire mbuf chain, [Jacobson 1988a] reports that it improves performance by increasing parallelism.

Transferring one mbuf at a time (up to 2048 bytes) allows the CPU to prepare a packet while the network hardware is transmitting. Contrast this to sending a large mbuf chain: while the chain is being constructed, the network and the receiving system are idle. On the system described in [Jacobson 1988a], this change resulted in a 20% increase in network throughput.

It is important to make sure the send buffer is always larger than the bandwidth-delay product of a connection (Section 20.7 of Volume 1). For example, if TCP discovers that the connection can hold 20 segments before an acknowledgment is received, the send buffer must be large enough to hold the 20 unacknowledged segments. If it is too small, TCP will run out of data to send before the first acknowledgment is returned and the connection will be idle for some period of time.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.8 read, readv, recvfrom, and recvmsg System Calls

These four system calls, which we refer to collectively as *read system calls*, receive data from a network connection. The first three system calls are simpler interfaces to the most general read system call, `recvmsg`. [Figure 16.27](#) summarizes the features of the four read system calls and one library function (`recv`).

**Figure 16.27. Read system calls.**

Function	Type of descriptor	Number of buffers	Return sender's address?	Flags?	Return control information?
read	any	1			
readv	any	[1..UIO_MAXIOV]			
recv	sockets only	1		•	
recvfrom	sockets only	1	:	•	
recvmsg	sockets only	[1..UIO_MAXIOV]			•

In Net/3, recv is implemented as a library function that calls recvfrom. For binary compatibility with previously compiled programs, the kernel maps the old recv system call to the function orecv. We discuss only the kernel implementation of recvfrom.

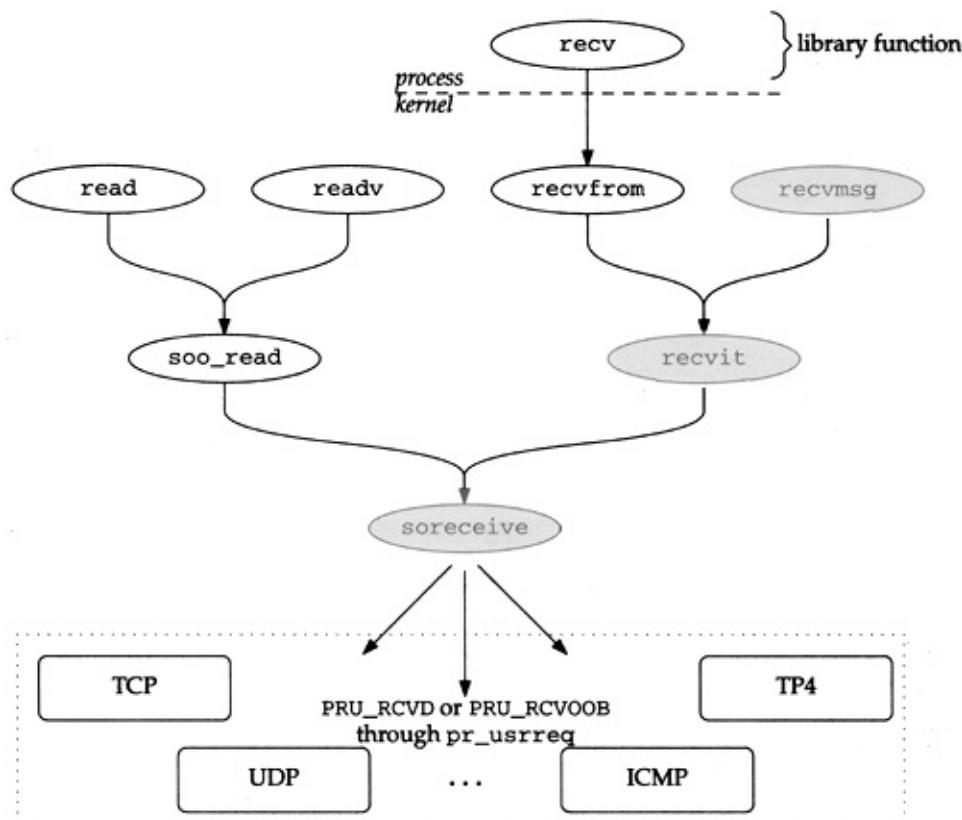
The read and readv system calls are valid with any descriptor, but the remaining calls are valid only with socket descriptors.

As with the write calls, multiple buffers are specified by an array of iovec structures. For datagram protocols, recvfrom and recvmsg return the source address associated with each incoming datagram. For connection-oriented protocols, getpeername returns the address associated with the other end of the connection. The flags associated with the receive calls are shown in [Section 16.11](#).

As with the write calls, the receive calls

utilize a common function, in this case `soreceive`, to do all the work. [Figure 16.28](#) illustrates the flow of control for the read system calls.

**Figure 16.28. All socket input is processed by `soreceive`.**



We discuss only the three shaded functions in [Figure 16.28](#). The remaining functions are left for readers to investigate on their own.

---

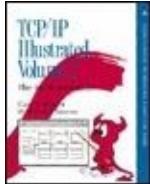
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.9 recvmsg System Call

The `recvmsg` function is the most general read system call. Addresses, control information, and receive flags may be discarded without notification if a process uses one of the other read system calls while this information is pending. [Figure 16.29](#) shows the `recvmsg` function.

**Figure 16.29. `recvmsg` system call.**

```

433 struct recvmsg_args {
434     int      s;
435     struct msghdr *msg;
436     int      flags;
437 };
438
439 recvmsg(p, uap, retval)
440 struct proc *p;
441 struct recvmsg_args *uap;
442 int      *retval;
443 {
444     struct msghdr msg;
445     struct iovec aiov[UIO_SMALLIOV], *uiov, *iov;
446     int      error;
447
448     if ((u_int) msg.msg_iovlen >= UIO_SMALLIOV) {
449         if ((u_int) msg.msg_iovlen >= UIO_MAXIOV)
450             return (EMSGSIZE);
451         MALLOC(iov, struct iovec *,
452                sizeof(struct iovec) * (u_int) msg.msg_iovlen, M_IOV,
453                M_WAITOK);
454     } else
455         iov = aiov;
456     msg.msg_flags = uap->flags;
457     uiov = msg.msg iov;
458     msg.msg iov = iov;
459     if (error = copyin((caddr_t) uiov, (caddr_t) iov,
460                        (unsigned) (msg.msg_iovlen * sizeof(struct iovec))))
461         goto done;
462     if ((error = recvit(p, uap->s, &msg, (caddr_t) 0, retval)) == 0) {
463         msg.msg iov = uiov;
464         error = copyout((caddr_t) &msg, (caddr_t) uap->msg, sizeof(msg));
465     }
466 done:
467     if (iov != aiov)
468         FREE(iov, M_IOV);
469     return (error);
470 }

```

uipc\_syscalls.c

## 433-445

The three arguments to recvmsg are: the socket descriptor; a pointer to a msghdr structure; and several control flags.

## Copy iov array

## 446-461

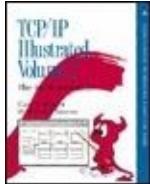
As with sendmsg, recvmsg copies the msghdr structure into the kernel, allocates a larger iovec array if the automatic array aiov is too small, and copies the array entries from the process into the kernel array pointed to by iov (Section 16.4). The flags provided as the third argument are copied into the msghdr structure.

## recvit and cleanup

### 462-470

After recvit has received data, the msghdr structure is copied back into the process with the updated buffer lengths and flags. If a larger iovec structure was allocated, it is released before recvmsg returns.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.10 recvit Function

The recvit function shown in [Figures 16.30](#) and [16.31](#) is called from recv, recvfrom, and recvmsg. It prepares a uio structure for processing by soreceive based on the msghdr structure prepared by the recvxxx calls.

**Figure 16.30. recvit function: initialize uio structure.**

```
471 recvit(p, s, mp, namelenp, rtsize)                                uipc_syscalls.c
472 struct proc *p;
473 int     s;
474 struct msghdr *mp;
475 caddr_t namelenp;
476 int     *rtsize;
477 {
478     struct file *fp;
479     struct uio auio;
480     struct iovec *iov;
481     int     i;
482     int     len, error;
483     struct mbuf *from = 0, *control = 0;
484     if (error = getsock(p->p_fd, s, &fp))
485         return (error);
486     auio.uio_iov = mp->msg_iov;
487     auio.uio_iovcnt = mp->msg_iovlen;
488     auio.uio_segflg = UIO_USERSPACE;
489     auio.uio_rw = UIO_READ;
490     auio.uio_procp = p;
491     auio.uio_offset = 0;          /* XXX */
492     auio.uio_resid = 0;
493     iov = mp->msg_iov;
494     for (i = 0; i < mp->msg_iovlen; i++, iov++) {
495         if (iov->iov_len < 0)
496             return (EINVAL);
497         if ((auio.uio_resid += iov->iov_len) < 0)
498             return (EINVAL);
499     }
500     len = auio.uio_resid;
```

**Figure 16.31. recvit function: return results.**

---

```

501     if (error = soreceive((struct socket *) fp->f_data, &from, &auio,
502         (struct mbuf **) 0, mp->msg_control ? &control : (struct mbuf **) 0,
503             &mp->msg_flags)) {
504         if (auio.uio_resid != len && (error == ERESTART ||
505             error == EINTR || error == EWOULDBLOCK))
506             error = 0;
507     }
508     if (error)
509         goto out;
510     *retsize = len - auio.uio_resid;
511     if (mp->msg_name) {
512         len = mp->msg_namelen;
513         if (len <= 0 || from == 0)
514             len = 0;
515         else {
516             if (len > from->m_len)
517                 len = from->m_len;
518             /* else if len < from->m_len ??? */
519             if (error = copyout(mtod(from, caddr_t),
520                 (caddr_t) mp->msg_name, (unsigned) len))
521                 goto out;
522         }
523         mp->msg_namelen = len;
524         if (namelenp &&
525             (error = copyout((caddr_t) &len, namelenp, sizeof(int)))) {
526             goto out;
527         }
528     }
529     if (mp->msg_control) {
530         len = mp->msg_controllen;
531         if (len <= 0 || control == 0)
532             len = 0;
533         else {
534             if (len >= control->m_len)
535                 len = control->m_len;
536             else
537                 mp->msg_flags |= MSG_CTRUNC;
538             error = copyout((caddr_t) mtod(control, caddr_t),
539                 (caddr_t) mp->msg_control, (unsigned) len);
540         }
541         mp->msg_controllen = len;
542     }
543     out:
544     if (from)
545         m_freem(from);
546     if (control)
547         m_freem(control);
548     return (error);
549 }
```

---

471-500

getsock returns the file structure for the descriptor s, and then recvit initializes the uio structure to describe a read transfer from the kernel to the process. The number of bytes to transfer is computed

by summing the msg iovlen members of the iovec array. The total is saved in uio\_resid and in len.

The second half of recvit, shown in [Figure 16.31](#), calls soreceive and copies the results back to the process.

## Call soreceive

501-510

soreceive implements the complex semantics of receiving data from the socket buffers. The number of bytes transferred is saved in \*retsize and returned to the process. When a signal arrives or a blocking condition occurs after some data has been copied to the process (len is not equal to uio\_resid), the error is discarded and the partial transfer is reported.

## Copy address and control information to the process

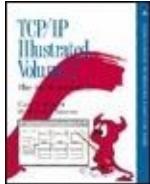
511-542

If the process provided a buffer for an address or control information or both, the buffers are filled and their lengths adjusted according to what soreceive returned. An address may be truncated if the buffer is too small. This can be detected by the process if it saves the buffer length before the read call and compares it with the value returned by the kernel in the namelenp variable (or in the length field of the sockaddr structure). Truncation of control information is reported by setting MSG\_CTRUNC in msg\_flags. See also [Exercise 16.7](#).

## Cleanup

### 543-549

At out, the mbufs allocated for the source address and the control information are released.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.11 soreceive Function

This function transfers data from the receive buffer of the socket to the buffers specified by the process. Some protocols provide an address specifying the sender of the data, and this can be returned along with additional control information that may be present. Before examining the code, we need to discuss the semantics of a receive operation, out-of-band data, and the organization of a socket's receive buffer.

Figure 16.32 lists the flags that are recognized by the kernel during soreceive.

**Figure 16.32. recvxxx system calls: flag**

## values passed to kernel.

flags	Description	Reference
<i>MSG_DONTWAIT</i>	do not wait for resources during this call	Figure 16.38
<i>MSG_OOB</i>	receive out-of-band data instead of regular data	Figure 16.39
<i>MSG_PEEK</i>	receive a copy of the data without consuming it	Figure 16.43
<i>MSG_WAITALL</i>	wait for data to fill buffers before returning	Figure 16.50

`recvmsg` is the only read system call that returns flags to the process. In the other calls, the information is discarded by the kernel before control returns to the process. [Figure 16.33](#) lists the flags that `recvmsg` can set in the `msghdr` structure.

**Figure 16.33. `recvmsg` system call: `msg_flag` values returned by kernel.**

msg_flags	Description	Reference
<i>MSG_CTRUNC</i>	the control information received was larger than the buffer provided	Figure 16.31
<i>MSG_EOR</i>	the data received marks the end of a logical record	Figure 16.48
<i>MSG_OOB</i>	the buffer(s) contains out-of-band data	Figure 16.45
<i>MSG_TRUNC</i>	the message received was larger than the buffer(s) provided	Figure 16.51

## Out-of-Band Data

Out-of-band (OOB) data semantics vary widely among protocols. In general, protocols expedite OOB data along a previously established communication link.

The OOB data might not remain in sequence with previously sent regular data. The socket layer supports two mechanisms to facilitate handling OOB data in a protocol-independent way: tagging and synchronization. In this chapter we describe the abstract OOB mechanisms implemented by the socket layer. UDP does not support OOB data. The relationship between TCP's urgent data mechanism and the socket OOB mechanism is described in the TCP chapters.

A sending process tags data as OOB data by setting the `MSG_OOB` flag in any of the `sendxxx` calls, `sosend` passes this information to the socket's protocol, which provides any special services, such as expediting the data or using an alternate queueing strategy.

When a protocol receives OOB data, the data is set aside instead of placing it in the socket's receive buffer. A process receives the pending OOB data by setting the `MSG_OOB` flag in one of the `recvxxx` calls. Alternatively, the receiving process can ask

the protocol to place OOB data inline with the regular data by setting the SO\_OOBINLINE socket option ([Section 17.3](#)). When SO\_OOBINLINE is set, the protocol places incoming OOB data in the receive buffer with the regular data. In this case, MSG\_OOB is not used to receive the OOB data. Read calls return either all regular data or all OOB data. The two types are never mixed in the input buffers of a single input system call. A process that uses recvmsg to receive data can examine the MSG\_OOB flag to determine if the returned data is regular data or OOB data that has been placed inline.

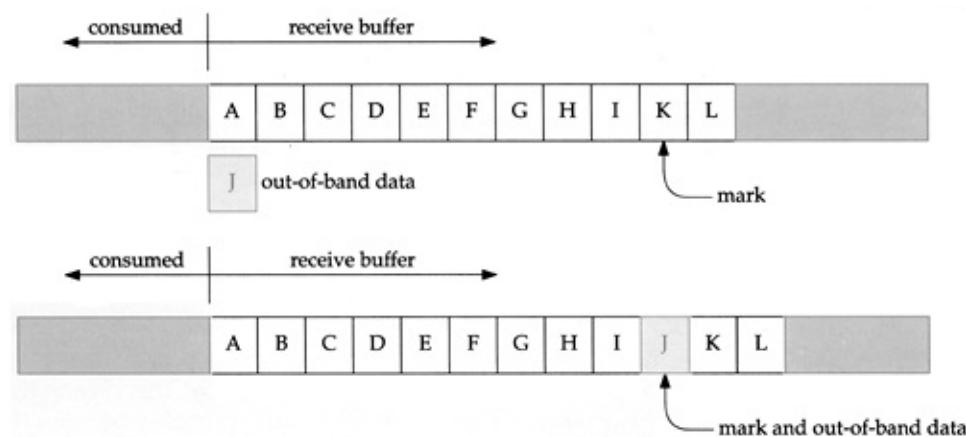
The socket layer supports synchronization of OOB and regular data by allowing the protocol layer to mark the point in the regular data stream at which OOB data was received. The receiver can determine when it has reached this mark by using the SIOCATMARK ioctl command after each read system call. When receiving regular data, the socket layer ensures that only the bytes preceding the mark are returned in a single message so that the receiver does not inadvertently pass the

mark. If additional OOB data is received before the receiver reaches the mark, the mark is silently advanced.

## Example

Figure 16.34 illustrates the two methods of receiving out-of-band data. In both examples, bytes A through I have been received as regular data, byte J as out-of-band data, and bytes K and L as regular data. The receiving process has accepted all data up to but not including byte A.

**Figure 16.34. Receiving out-of-band data.**



In the first example, the process can read bytes A through I or, if MSG\_OOB is set,

byte J. Even if the length of the read request is more than 9 bytes (AI), the socket layer returns only 9 bytes to avoid passing the out-of-band synchronization mark. When byte I is consumed, SIOCATMARK is true; it is not necessary to consume byte J for the process to reach the out-of-band mark.

In the second example, the process can read only bytes A through I, at which point SIOCATMARK is true. A second call can read bytes J through L.

In [Figure 16.34](#), byte J is *not* the byte identified by TCP's urgent pointer. The urgent pointer in this example would point to byte K. See [Section 29.7](#) for details.

## Other Receive Options

A process can set the MSG\_PEEK flag to retrieve data without consuming it. The data remains on the receive queue until a read system call without MSG\_PEEK is processed.

The `MSG_WAITALL` flag indicates that the call should not return until enough data can be returned to fulfill the entire request. Even if `soreceive` has some data that can be returned to the process, it waits until additional data has been received.

When `MSG_WAITALL` is set, `soreceive` can return without filling the buffer in the following cases:

- the read-half of the connection is closed,
- the socket's receive buffer is smaller than the size of the read,
- an error occurs while the process is waiting for additional data,
- out-of-band data becomes available, or
- the end of a logical record occurs before the read buffer is filled.

NFS is the only software in Net/3 that uses the `MSG_WAITALL` and `MSG_DONTWAIT` flags.

`MSG_DONTWAIT` can be set by a process to issue a nonblocking read system call without selecting nonblocking I/O with `ioctl` or `fcntl`.

## Receive Buffer Organization: Message Boundaries

For protocols that support message boundaries, each message is stored in a single chain of mbufs. Multiple messages in the receive buffer are linked together by `m_nextpkt` to form a queue of mbufs ([Figure 2.21](#)). The protocol processing layer adds data to the receive queue and the socket layer removes data from the receive queue. The high-water mark for a receive buffer restricts the amount of data that can be stored in the buffer.

When `PR_ATOMIC` is not set, the protocol layer stores as much data in the buffer as possible and discards the portion of the incoming data that does not fit. For TCP, this means that any data that arrives and is outside the receive window is discarded. When `PR_ATOMIC` is set, the entire

message must fit within the buffer. If the message does not fit, the protocol layer discards the entire message. For UDP, this means that incoming datagrams are discarded when the receive buffer is full, probably because the process is not reading datagrams fast enough.

Protocols with PR\_ADDR set use sbappendaddr to construct an mbuf chain and add it to the receive queue. The chain contains an mbuf with the source address of the message, 0 or more control mbufs, followed by 0 or more mbufs containing the data.

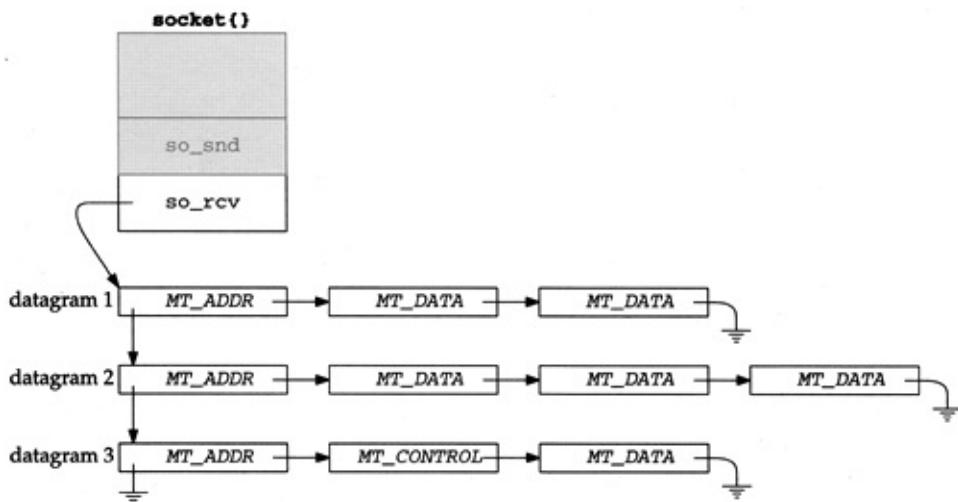
For SOCK\_SEQPACKET and SOCK\_RDM protocols, the protocol builds an mbuf chain for each record and calls sbappendrecord to append the record to the end of the receive buffer if PR\_ATOMIC is set. If PR\_ATOMIC is not set (OSI's TP4), a new record is started with sbappendrecord. Additional data is added to the record with sbappend.

It is not correct to assume that PR\_ATOMIC indicates the buffer

organization. For example, TP4 does not have PR\_ATOMIC set, but supports record boundaries with the M\_EOR flag.

[Figure 16.35](#) illustrates the organization of a UDP receive buffer consisting of 3 mbuf chains (i.e., three datagrams). The m\_type value for each mbuf is included.

**Figure 16.35. UDP receive buffer consisting of three datagrams.**



In the figure, the third datagram has some control information associated with it. Three UDP socket options can cause control information to be placed in the receive buffer. See [Figure 22.5](#) and [Section](#)

[23.7](#) for details.

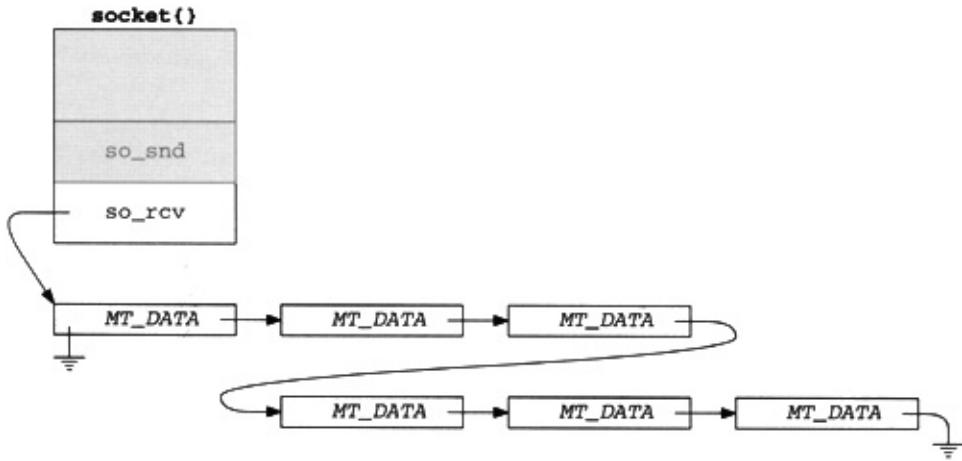
For PR\_ATOMIC protocols, sb\_lowat is ignored while data is being received. When PR\_ATOMIC is not set, sb\_lowat is the smallest number of bytes returned in a read system call. There are some exceptions to this rule, discussed with [Figure 16.41](#).

## Receive Buffer Organization: No Message Boundaries

When the protocol does not maintain message boundaries (i.e., SOCK\_STREAM protocols such as TCP), incoming data is appended to the end of the last mbuf chain in the buffer with sbappend. Incoming data is trimmed to fit within the receive buffer, and sb\_lowat puts a lower bound on the number of bytes returned by a read system call.

[Figure 16.36](#) illustrates the organization of a TCP receive buffer, which contains only regular data.

**Figure 16.36. so\_rcv buffer for TCP.**

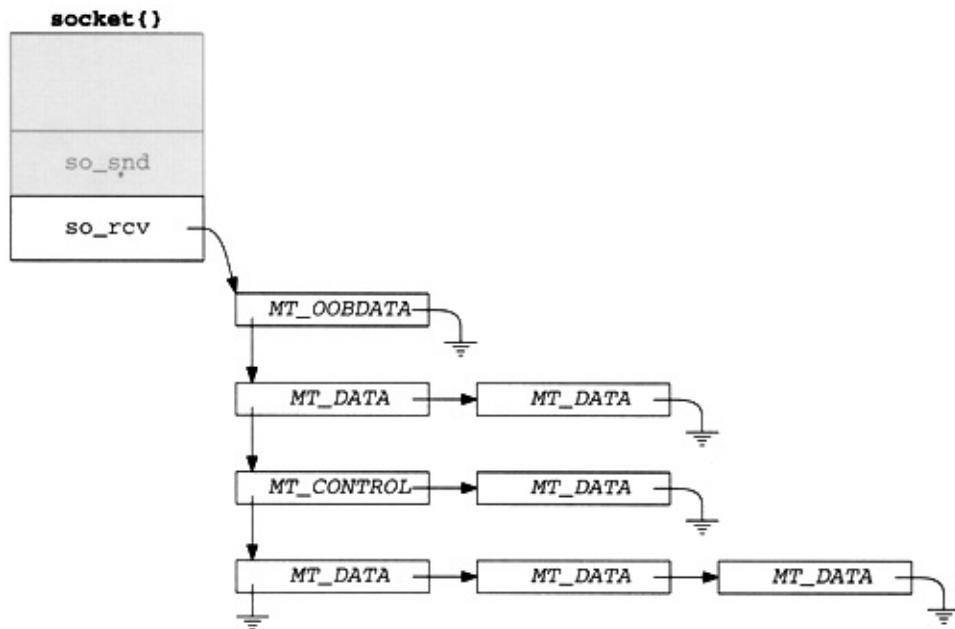


## Control Information and Out-of-band Data

Unlike TCP, some stream protocols support control information and call `sbappendcontrol` to append the control information and the associated data as a new mbuf chain in the receive buffer. If the protocol supports inline OOB data, `sbinserootob` inserts a new mbuf chain just after any mbuf chain that contains OOB data, but before any mbuf chain with regular data. This ensures that incoming OOB data is queued ahead of any regular data.

Figure 16.37 illustrates the organization of a receive buffer that contains control information and OOB data.

**Figure 16.37. so\_rcv buffer with control and OOB data.**



The Unix domain stream protocol supports control information and the OSI TP4 protocol supports MT\_OOBDATA mbufs. TCP does not support control data nor does it support the MT\_OOBDATA form of out-of-band data. If the byte identified by TCP's urgent pointer is stored inline (SO\_OOBINLINE is set), it appears as

regular data, not OOB data. TCP's handling of the urgent pointer and the associated byte is described in [Section 29.7](#).

---

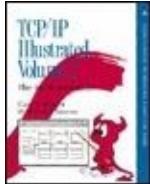
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.12 soreceive Code

We now have enough background information to discuss soreceive in detail. While receiving data, soreceive must respect message boundaries, handle addresses and control information, and handle any special semantics identified by the read flags (Figure 16.32). The general rule is that soreceive processes one record per call and tries to return the number of bytes requested. Figure 16.38 shows an overview of the function.

**Figure 16.38. soreceive function: overview.**

```
439 soreceive(so, paddr, uio, mp0, controlp, flagsp)
440 struct socket *so;
441 struct mbuf **paddr;
442 struct uio *uio;
443 struct mbuf **mp0;
444 struct mbuf **controlp;
445 int     *flagsp;
446 {
447     struct mbuf *m, **mp;
448     int      flags, len, error, s, offset;
449     struct protosw *pr = so->so_proto;
450     struct mbuf *nextrecord;
451     int      moff, type;
452     int      orig_resid = uio->uio_resid;

453     mp = mp0;
454     if (paddr)
455         *paddr = 0;
456     if (controlp)
457         *controlp = 0;
458     if (flagsp)
459         flags = *flagsp & ~MSG_EOR;
460     else
461         flags = 0;

        /* MSG_OOB processing and */
        /* implicit connection confirmation */

483     restart:
484     if (error = sblock(&so->so_rcv, SBLOCKWAIT(flags)))
485         return (error);
486     s = splnet();
487     m = so->so_rcv.sb_mb;

        /* if necessary, wait for data to arrive */

542     dontblock:
543     if (uio->uio_procp)
544         uio->uio_procp->p_stats->p_ru.ru_msgrcv++;
545     nextrecord = m->m_nextpkt;

        /* process address and control information */

591     if (m) {
592         if ((flags & MSG_PEEK) == 0)
593             m->m_nextpkt = nextrecord;
594         type = m->m_type;
595         if (type == MT_OOBDATA)
596             flags |= MSG_OOB;
597     }
```

```
    /* process data */

693     )          /* while more data and more space to fill */

    /* cleanup */

715     release:
716     sbunlock(&so->so_rcv);
717     splx(s);
718     return (error);
719 }
```

————— uipc\_socket.c

## 439-446

soreceive has six arguments. so is a pointer to the socket. A pointer to an mbuf to receive address information is returned in \*paddr. If mp0 points to an mbuf pointer, soreceive transfers the receive buffer data to an mbuf chain pointed to by \*mp0. In this case, the uio structure is used only for the count in uio\_resid. If mp0 is null, soreceive copies the data into buffers described by the uio structure. A pointer to the mbuf containing control information is returned in \*controlp, and soreceive returns the flags described in [Figure 16.33](#) in \*flagsp.

## 447-453

soreceive starts by setting pr to point to the socket's protocol switch structure and

saving uio\_resid (the size of the receive request) in orig\_resid. If control information or addressing information is copied from the kernel to the process, orig\_resid is set to 0. If data is copied, uio\_resid is updated. In either case, orig\_resid will not equal uio\_resid. This fact is used at the end of soreceive ([Figure 16.51](#)).

454-461

\*paddr and \*controlp are cleared. The flags passed to soreceive in \*flagsp are saved in flags after the MSG\_EOR flag is cleared ([Exercise 16.8](#)). flagsp is a value-result argument, but only the recvmsg system call can receive the result flags. If flagsp is null, flags is set to 0.

483-487

Before accessing the receive buffer, sblock locks the buffer. soreceive waits for the lock unless MSG\_DONTWAIT is set in flags.

This is another side effect of supporting calls to the socket layer from NFS

within the kernel.

Protocol processing is suspended, so soreceive is not interrupted while it examines the buffer. m is the first mbuf on the first chain in the receive buffer.

## If necessary, wait for data

488-541

soreceive checks several conditions and if necessary waits for more data to arrive in the buffer before continuing. If soreceive sleeps in this code, it jumps back to restart when it wakes up to see if enough data has arrived. This continues until the request can be satisfied.

542-545

soreceive jumps to dontblock when it has enough data to satisfy the request. A pointer to the second chain in the receive buffer is saved in nextrecord.

## Process address and control information

546-590

Address information and control information are processed before any other data is transferred from the receive buffer.

## Setup data transfer

591-597

Since only OOB data or regular data is transferred in a single call to soreceive, this code remembers the type of data at the front of the queue so soreceive can stop the transfer when the type changes.

## Mbuf data transfer loop

598-692

This loop continues as long as there are mbufs in the buffer (m is not null), the requested number of bytes has not been transferred (uio\_resid > 0), and no error has occurred.

## Cleanup

693-719

The remaining code updates various pointers, flags, and offsets; releases the socket buffer lock; enables protocol processing; and returns.

In [Figure 16.39](#), soreceive handles requests for OOB data.

**Figure 16.39. soreceive function: out-of-band data.**

```
462     if (flags & MSG_OOB) {
463         m = m_get(M_WAIT, MT_DATA);
464         error = (*pr->pr_usrreq) (so, PRU_RCVOOB,
465             m, (struct mbuf *) (flags & MSG_PEEK), (struct mbuf *) 0);
466         if (error)
467             goto bad;
468         do {
469             error = uiomove(mtod(m, caddr_t),
470                             (int) min(uio->uio_resid, m->m_len), uio);
471             m = m_free(m);
472         } while (uio->uio_resid && error == 0 && m);
473     bad:
474         if (m)
475             m_freem(m);
476         return (error);
477     }
```

*uipc\_socket.c*

*uipc\_socket.c*

Receive OOB data

462-477

Since OOB data is not stored in the receive buffer, soreceive allocates a standard mbuf and issues the PRU\_RCVOOB request to the protocol. The while loop copies any data returned by the protocol to the buffers specified by uio. After the copy, soreceive returns 0 or the error code.

UDP always returns EOPNOTSUPP for the PRU\_RCVOOB request. See [Section 30.2](#) for details regarding TCP urgent processing. In [Figure 16.40](#), soreceive handles connection confirmation.

## Figure 16.40. soreceive function: connection confirmation.

```
478     if (mp)
479         *mp = (struct mbuf *) 0;
480     if (so->so_state & SS_ISCONFIRMING && uio->uio_resid)
481         (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
482                             (struct mbuf *) 0, (struct mbuf *) 0);
----- uipc_socket.c
```

## Connection confirmation

478-482

If the data is to be returned in an mbuf

chain, \*mp is initialized to null. If the socket is in the SO\_ISCONFIRMING state, the PRU\_RCVD request notifies the protocol that the process is attempting to receive data.

The SO\_ISCONFIRMING state is used only by the OSI stream protocol, TP4. In TP4, a connection is not considered complete until a user-level process has confirmed the connection by attempting to send or receive data. The process can reject a connection by calling shutdown or close, perhaps after calling getpeername to determine where the connection came from.

[Figure 16.38](#) showed that the receive buffer is locked before it is examined by the code in [Figure 16.41](#). This part of soreceive determines if the read system call can be satisfied by the data that is already in the receive buffer.

## **Figure 16.41. soreceive function: enough data?**

```

488     /*
489      * If we have less data than requested, block awaiting more
490      * (subject to any timeout) if:
491      *   1. the current count is less than the low water mark, or
492      *   2. MSG_WAITALL is set, and it is possible to do the entire
493      *      receive operation at once if we block (resid <= hiwat).
494      *   3. MSG_DONTWAIT is not set
495      *
496      * If MSG_WAITALL is set but resid is larger than the receive buffer,
497      * we have to do the receive in sections, and thus risk returning
498      * a short count if a timeout or signal occurs after we start.
499      */
500     if (m == 0 || ((flags & MSG_DONTWAIT) == 0 &&
501             so->so_rcv.sb_cc < uio->uio_resid) &&
502             (so->so_rcv.sb_cc < so->so_rcv.sb_lowat ||
503              ((flags & MSG_WAITALL) && uio->uio_resid <= so->so_rcv.sb_hiwat)) &&
504             m->m_nextpkt == 0 && (pr->pr_flags & PR_ATOMIC) == 0) {

```

uipc\_socket.c

## Can the call be satisfied now?

**488-504**

The general rule for soreceive is that it waits until enough data is in the receive buffer to satisfy the entire read. There are several conditions that cause an error or less data than was requested to be returned.

If any of the following conditions are true, the process is put to sleep to wait for more data to arrive so the call can be satisfied:

- There is no data in the receive buffer (m equals 0).
- There is not enough data to satisfy the

entire read ( $sb\_cc < uio\_resid$  and  $MSG\_DONTWAIT$  is not set), the minimum amount of data is *not* available ( $sb\_cc < sb\_lowat$ ), and more data can be appended to this chain when it arrives ( $m\_nextpkt$  is 0 and  $PR\_ATOMIC$  is *not* set).

- There is not enough data to satisfy the entire read, a minimum amount of data *is* available, data can be added to this chain, but  $MSG\_WAITALL$  indicates that `soreceive` should wait until the entire read can be satisfied.

If the conditions in the last case are met but the read is too large to be satisfied without blocking ( $uio\_resid > sb\_hiwat$ ), `soreceive` continues without waiting for more data.

If there is some data in the buffer and  $MSG\_DONTWAIT$  is set, `soreceive` does not wait for more data.

There are several reasons why waiting for more data may not be appropriate. In Figure 16.42, `soreceive` checks for these

conditions and returns, or waits for more data to arrive.

## Figure 16.42. soreceive function: wait for more data?

```
505     if (so->so_error) {
506         if (m)
507             goto dontblock;
508         error = so->so_error;
509         if ((flags & MSG_PEEK) == 0)
510             so->so_error = 0;
511         goto release;
512     }
513     if (so->so_state & SS_CANTRCVMORE) {
514         if (m)
515             goto dontblock;
516         else
517             goto release;
518     }
519     for (; m; m = m->m_next)
520         if (m->m_type == MT_OOBDATA || (m->m_flags & M_EOR)) {
521             m = so->so_rcv.sb_mb;
522             goto dontblock;
523         }
524     if ((so->so_state & (SS_ISCONNECTED | SS_ISCONNECTING)) == 0 &&
525         (so->so_proto->pr_flags & PR_CONNREQUIRED)) {
526         error = ENOTCONN;
527         goto release;
528     }
529     if (uio->uio_resid == 0)
530         goto release;
531     if ((so->so_state & SS_NBIO) || (flags & MSG_DONTWAIT)) {
532         error = EWOULDBLOCK;
533         goto release;
534     }
535     sbunlock(&so->so_rcv);
536     error = sbwait(&so->so_rcv);
537     splx(s);
538     if (error)
539         return (error);
540     goto restart;
541 }
```

## Wait for more data?

505-534

At this point, soreceive has determined that it must wait for additional data to arrive before the read can be satisfied. Before waiting it checks for several additional conditions:

- 505-512

If the socket is in an error state and *empty* (*m* is null), soreceive returns the error code. If there is an error and the receive buffer also contains data (*m* is nonnull), the data is returned and a subsequent read returns the error when there is no more data. If MSG\_PEEK is set, the error is not cleared, since a read system call with MSG\_PEEK set should not change the state of the socket.

- 513-518

If the read-half of the connection has been closed and data remains in the receive buffer, sosend does not wait and returns the data to the process (at dontblock). If the receive buffer is empty, soreceive jumps to release and

the read system call returns 0, which indicates that the read-half of the connection is closed.

- **519-523**

If the receive buffer contains out-of-band data or the end of a logical record, soreceive does not wait for additional data and jumps to dontblock.

- **524-528**

If the protocol requires a connection and it does not exist, ENOTCONN is posted and the function jumps to release.

- **529-534**

If the read is for 0 bytes or nonblocking semantics have been selected, the function jumps to release and returns 0 or EWOULDBLOCK, respectively.

**Yes, wait for more data**

**535-541**

soreceive has now determined that it must wait for more data, and that it is reasonable to do so (i.e., some data will arrive). The receive buffer is unlocked while the process sleeps in sbwait. If sbwait returns because of an error or a signal, soreceive returns the error; otherwise the function jumps to restart to determine if the read can be satisfied now that more data has arrived.

As in sosend, a process can enable a receive timer for sbwait with the SO\_RCVTIMEO socket option. If the timer expires before any data arrives, sbwait returns EWOULDBLOCK.

The effect of this timer is not what one would expect. Since the timer gets reset every time there is activity on the socket buffer, the timer never expires if at least 1 byte arrives within the timeout interval. This can delay the return of the read system call for more than the value of the timer. sb\_timeo is an inactivity timer and does not put an upper bound on the amount of time that may be required to satisfy the read

system call.

At this point, soreceive is prepared to transfer some data from the receive buffer. [Figure 16.43](#) shows the transfer of any address information.

### Figure 16.43. soreceive function: return address information.

```
542     dontblock:  
543         if (uio->uio_procp)  
544             uio->uio_procp->p_stats->p_ru.ru_msgrcv++;  
545         nextrecord = m->m_nextpkt;  
546         if (pr->pr_flags & PR_ADDR) {  
547             orig_resid = 0;  
548             if (flags & MSG_PEEK) {  
549                 if (paddr)  
550                     *paddr = m_copy(m, 0, m->m_len);  
551                     m = m->m_next;  
552             } else {  
553                 sbfree(&so->so_rcv, m);  
554                 if (paddr) {  
555                     *paddr = m;  
556                     so->so_rcv.sb_mb = m->m_next;  
557                     m->m_next = 0;  
558                     m = so->so_rcv.sb_mb;  
559                 } else {  
560                     MFREE(m, so->so_rcv.sb_mb);  
561                     m = so->so_rcv.sb_mb;  
562                 }  
563             }  
564         }
```

uipc\_socket.c

**dontblock**

542-545

nextrecord maintains a reference to the

next record that appears in the receive buffer. This is used at the end of soreceive to attach the remaining mbufs to the socket buffer after the first chain has been discarded.

## Return address information

546-564

If the protocol provides addresses, such as UDP, the mbuf containing the address is removed from the mbuf chain and returned in \*paddr. If paddr is null, the address is discarded.

Throughout soreceive, if MSG\_PEEK is set, the data is not removed from the buffer.

The code in [Figure 16.44](#) processes any control mbufs that are in the buffer.

**Figure 16.44. soreceive function: control information.**

```

565     while (m && m->m_type == MT_CONTROL && error == 0) {
566         if (flags & MSG_PEEK) {
567             if (controlp)
568                 *controlp = m_copy(m, 0, m->m_len);
569             m = m->m_next;
570         } else {
571             sbfree(&so->so_rcv, m);
572             if (controlp) {
573                 if (pr->pr_domain->dom_externalize &&
574                     mtod(m, struct cmsghdr *)->cmsg_type ==
575                     SCM_RIGHTS)
576                     error = (*pr->pr_domain->dom_externalize) (m);
577                 *controlp = m;
578                 so->so_rcv.sb_mb = m->m_next;
579                 m->m_next = 0;
580                 m = so->so_rcv.sb_mb;
581             } else {
582                 MFREE(m, so->so_rcv.sb_mb);
583                 m = so->so_rcv.sb_mb;
584             }
585         }
586         if (controlp) {
587             orig_resid = 0;
588             controlp = &(*controlp)->m_next;
589         }
590     }

```

uipc\_socket.c

## Return control information

565-590

Each control mbuf is removed from the buffer (or copied if MSG\_PEEK is set) and attached to \*controlp. If controlp is null, the control information is discarded.

If the process is prepared to receive control information, the protocol has a dom\_externalize function defined, and if the control mbuf contains a SCM\_RIGHTS (access rights) message, the dom\_externalize function is called. This

function takes any kernel action associated with receiving the access rights. Only the Unix protocol domain supports access rights, as discussed in [Section 7.3](#). If the process is not prepared to receive control information (controlp is null) the mbuf is discarded.

The loop continues while there are more mbufs with control information and no error has occurred.

For the Unix protocol domain, the `dom_externalize` function implements the semantics of passing file descriptors by modifying the file descriptor table of the receiving process.

After the control mbufs are processed, `m` points to the next mbuf on the chain. If the chain does not contain any mbufs after the address, or after the control information, `m` is null. This occurs, for example, when a 0-length UDP datagram is queued in the receive buffer. In [Figure 16.45](#) `soreceive` prepares to transfer the data from the mbuf chain.

## Figure 16.45. soreceive function: mbuf transfer setup.

```
591     if (m) {
592         if ((flags & MSG_PEEK) == 0)
593             m->m_nextpkt = nextrecord;
594         type = m->m_type;
595         if (type == MT_OOBDATA)
596             flags |= MSG_OOB;
597     }
```

### Prepare to transfer data

591-597

After the control mbufs have been processed, the chain should contain regular, out-of-band data mbufs or no mbufs at all. If m is null, soreceive is finished with this chain and control drops to the bottom of the while loop. If m is not null, any remaining chains (nextrecord) are reattached to m and the type of the next mbuf is saved in type. If the next mbuf contains OOB data, MSG\_OOB is set in flags, which is later returned to the process. Since TCP does not support the MT\_OOBDATA form of out-of-band data, MSG\_OOB will never be returned for reads on TCP sockets.

[Figure 16.47](#) shows the first part of the mbuf transfer loop. [Figure 16.46](#) lists the variables updated within the loop.

## **Figure 16.46. soreceive function: loop variables.**

Variable	Description
moff	the offset of the next byte to transfer when MSG_PEEK is set
offset	the offset of the OOB mark when MSG_PEEK is set
uio_resid	the number of bytes remaining to be transferred
len	the number of bytes to be transferred from this mbuf; may be less than m_len if uio_resid is small, or if the OOB mark is near

## **Figure 16.47. soreceive function: uiomove.**

---

```

598     moff = 0;
599     offset = 0;
600     while (m && uio->uio_resid > 0 && error == 0) {
601         if (m->m_type == MT_OOBDATA) {
602             if (type != MT_OOBDATA)
603                 break;
604         } else if (type == MT_OOBDATA)
605             break;
606         so->so_state &= ~SS_RCVATM;
607         len = uio->uio_resid;
608         if (so->so_oobmark && len > so->so_oobmark - offset)
609             len = so->so_oobmark - offset;
610         if (len > m->m_len - moff)
611             len = m->m_len - moff;
612         /*
613          * If mp is set, just pass back the mbufs.
614          * Otherwise copy them out via the uio, then free.
615          * Sockbuf must be consistent here (points to current mbuf,
616          * it points to next record) when we drop priority;
617          * we must note any additions to the sockbuf when we
618          * block interrupts again.
619          */
620         if (mp == 0) {
621             splx(s);
622             error = uiomove(mtod(m, caddr_t) + moff, (int) len, uio);
623             s = splnet();
624         } else
625             uio->uio_resid -= len;

```

---

uipc\_socket.c

## 598-600

During each iteration of the while loop, the data in a single mbuf is transferred to the output chain or to the uio buffers. The loop continues while there are more mbufs, the process's buffers are not full, and no error has occurred.

## Check for transition between OOB and regular data

## 600-605

If, while processing the mbuf chain, the

type of the mbuf changes, the transfer stops. This ensures that regular and out-of-band data are not both returned in the same message. This check does not apply to TCP.

## Update OOB mark

606-611

The distance to the oobmark is computed and limits the size of the transfer, so the byte before the mark is the last byte transferred. The size of the transfer is also limited by the size of the mbuf. This code does apply to TCP.

612-625

If the data is being returned to the uio buffers, uiomove is called. If the data is being returned as an mbuf chain, uio\_resid is adjusted to reflect the number of bytes moved.

To avoid suspending protocol processing for a long time, protocol processing is enabled during the call to uiomove.

Additional data may appear in the receive buffer because of protocol processing while uiomove is running.

The code in [Figure 16.48](#) adjusts all the pointers and offsets to prepare for the next mbuf.

## Figure 16.48. soreceive function: update buffer.

```
626     if (len == m->m_len - moff) {
627         if (m->m_flags & M_EOR)
628             flags |= MSG_EOR;
629         if (flags & MSG_PEEK) {
630             m = m->m_next;
631             moff = 0;
632         } else {
633             nextrecord = m->m_nextpkt;
634             sbfree(&so->so_rcv, m);
635             if (mp) {
636                 *mp = m;
637                 mp = &m->m_next;
638                 so->so_rcv.sb_mb = m = m->m_next;
639                 *mp = (struct mbuf *) 0;
640             } else {
641                 MFREE(m, so->so_rcv.sb_mb);
642                 m = so->so_rcv.sb_mb;
643             }
644             if (m)
645                 m->m_nextpkt = nextrecord;
646         }
647     } else {
648         if (flags & MSG_PEEK)
649             moff += len;
650         else {
651             if (mp)
652                 *mp = m_copym(m, 0, len, M_WAIT);
653             m->m_data += len;
654             m->m_len -= len;
655             so->so_rcv.sb_cc -= len;
656         }
657     }

```

Finished with mbuf?

626-646

If all the bytes in the mbuf have been transferred, the mbuf must be discarded or the pointers advanced. If the mbuf contained the end of a logical record, MSG\_EOR is set. If MSG\_PEEK is set, soreceive skips to the next buffer. If MSG\_PEEK is not set, the buffer is discarded if the data was copied by uiomove, or appended to mp if the data is being returned in an mbuf chain.

## More data to process

647-657

There may be more data to process in the mbuf if the request didn't consume all the data, if so\_oobmark cut the request short, or if additional data arrived during uiomove. If MSG\_PEEK is set, moff is updated. If the data is to be returned on an mbuf chain, len bytes are copied and attached to the chain. The mbuf pointers and the receive buffer byte count are updated by the amount of data that was

transferred.

Figure 16.49 contains the code that handles the OOB offset and the MSG\_EOR processing.

### Figure 16.49. soreceive function: out-of-band data mark.

```
658     if (so->so_oobmark) {
659         if ((flags & MSG_PEEK) == 0) {
660             so->so_oobmark -= len;
661             if (so->so_oobmark == 0) {
662                 so->so_state |= SS_RCVATMARK;
663                 break;
664             }
665         } else {
666             offset += len;
667             if (offset == so->so_oobmark)
668                 break;
669         }
670     }
671     if (flags & MSG_EOR)
672         break;
```

### Update OOB mark

658-670

If the out-of-band mark is nonzero, it is decremented by the number of bytes transferred. If the mark has been reached, SS\_RCVATMARK is set and soreceive breaks out of the while loop. If MSG\_PEEK

is set, offset is updated instead of so\_oobmark.

## End of logical record

671-672

If the end of a logical record has been reached, soreceive breaks out of the mbuf processing loop so data from the next logical record is not returned with this message.

The loop in [Figure 16.50](#) waits for more data to arrive when MSG\_WAITALL is set and the request is not complete.

**Figure 16.50. soreceive function:  
MSG\_WAITALL processing.**

```

673     /*
674      * If the MSG_WAITALL flag is set (for non-atomic socket),
675      * we must not quit until "uio->uio_resid == 0" or an error
676      * termination.  If a signal/timeout occurs, return
677      * with a short count but without error.
678      * Keep sockbuf locked against other readers.
679      */
680     while (flags & MSG_WAITALL && m == 0 && uio->uio_resid > 0 &&
681           !sosendallatonce(so) && !nextrecord) {
682         if (so->so_error || so->so_state & SS_CANTRCVMORE)
683             break;
684         error = sbwait(&so->so_rcv);
685         if (error) {
686             sbunlock(&so->so_rcv);
687             splx(s);
688             return (0);
689         }
690         if (m == so->so_rcv.sb_mb)
691             nextrecord = m->m_nextpkt;
692     }
693 }                                     /* while more data and more space to fill */

```

---

## MSG\_WAITALL

673-681

If `MSG_WAITALL` is set, there is no more data in the receive buffer (`m` equals 0), the caller wants more data, `sosendallatonce` is false, and this is the last record in the receive buffer (`nextrecord` is null), then `soreceive` must wait for additional data.

**Error or no more data will arrive**

682-683

If an error is pending or the connection is

closed, the loop is terminated.

## **Wait for data to arrive**

684-689

sbswait returns when the receive buffer is changed by the protocol layer. If the wait was interrupted by a signal (error is nonzero), sosend returns immediately.

## **Synchronize m and nextrecord with receive buffer**

690-692

m and nextrecord are updated, since the receive buffer has been modified by the protocol layer. If data arrived in the mbuf, m will be nonzero and the while loop terminates.

## **Process next mbuf**

693

This is the end of the mbuf processing loop. Control returns to the loop starting on line 600 ([Figure 16.47](#)). As long as there is data in the receive buffer, more space to fill, and no error has occurred, the loop continues.

When soreceive stops copying data, the code in [Figure 16.51](#) is executed.

## Figure 16.51. soreceive function: cleanup.

```
694     if (m && pr->pr_flags & PR_ATOMIC) {                                uipc_socket.c
695         flags |= MSG_TRUNC;
696         if ((flags & MSG_PEEK) == 0)
697             (void) sbdroprecord(&so->so_rcv);
698     }
699     if ((flags & MSG_PEEK) == 0) {
700         if (m == 0)
701             so->so_rcv.sb_mb = nextrecord;
702         if (pr->pr_flags & PR_WANTRCVD && so->so_pcb)
703             (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
704                                 (struct mbuf *) flags, (struct mbuf *) 0,
705                                 (struct mbuf *) 0);
706     }
707     if (orig_resid == uio->uio_resid && orig_resid &&
708         (flags & MSG_EOR) == 0 && (so->so_state & SS_CANTRCVMORE) == 0) {
709         sbunlock(&so->so_rcv);
710         splx(s);
711         goto restart;
712     }
713     if (flagsp)
714         *flagsp |= flags;
```

## Truncated message

694-698

If the process received a partial message (a datagram or a record) because its receive buffer was too small, the process is notified by setting MSG\_TRUNC and the remainder of the message is discarded. MSG\_TRUNC (as with all receive flags) is available only to a process through the recvmsg system call, even though soreceive always sets the flags.

## End of record processing

699-706

If MSG\_PEEK is not set, the next mbuf chain is attached to the receive buffer and, if required, the protocol is notified that the receive operation has been completed by issuing the PRU\_RCVD protocol request. TCP uses this feature to update the receive window for the connection.

## Nothing transferred

707-712

If soreceive runs to completion, no data is

transferred, the end of a record is not reached, and the read-half of the connection is still active, then the buffer is unlocked and soreceive jumps back to restart to continue waiting for data.

713-714

Any flags set during soreceive are returned in \*flagsp, the buffer is unlocked, and soreceive returns.

## Analysis

soreceive is a complex function. Much of the complication is because of the intricate manipulation of pointers and the multiple types of data (out-of-band, address, control, regular) and multiple destinations (process buffers, mbuf chain).

Similar to sosend, soreceive has collected features over the years. A specialized receive function for each protocol would blur the boundary between the socket layer and the protocol layer, but it would simplify the code considerably.

[Partridge and Pink 1993] describe the creation of a custom soreceive function for UDP to checksum datagrams while they are copied from the receive buffer to the process. They note that modifying the generic soreceive function to support this feature would "make the already complicated socket routines even more complex."

---

[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)

## Chapter 16. Socket I/O

### 16.13 select System Call

In the following discussion we assume that the operation and semantics of select. For a detailed interface to select see [[Stevens 1992](#)].

[Figure 16.52](#) shows the conditions detected by

**Figure 16.52. select system call**

Description	Detected by selecting for:		
	reading	writing	exceptions
data available for reading	•		
read-half of connection is closed	•		
listen socket has queued connection	•		
socket error is pending	•		
space available for writing and a connection exists or is not required		•	
write-half of connection is closed		•	
socket error is pending		•	
OOB synchronization mark is pending			•

We start with the first half of the select system

## Figure 16.53. select function

```
-----sys_generic.c-----  
390 struct select_args {  
391     u_int    nd;  
392     fd_set *in, *ou, *ex;  
393     struct timeval *tv;  
394 };  
395 select(p, uap, retval)  
396 struct proc *p;  
397 struct select_args *uap;  
398 int    *retval;  
399 {  
400     fd_set ibits[3], obits[3];  
401     struct timeval atv;  
402     int      s, ncoll, error = 0, timo;  
403     u_int    ni;  
404     bzero((caddr_t) ibits, sizeof(ibits));  
405     bzero((caddr_t) obits, sizeof(obits));  
406     if (uap->nd > FD_SETSIZE)  
407         return (EINVAL);  
408     if (uap->nd > p->p_fd->fd_nfiles)  
409         uap->nd = p->p_fd->fd_nfiles; /* forgiving; slightly wrong */  
410     ni = howmany(uap->nd, NFDBITS) * sizeof(fd_mask);  
411 #define getbits(name, x) \  
412     if (uap->name && \  
413         (error = copyin((caddr_t)uap->name, (caddr_t)&ibits[x], ni))) \  
414         goto done;  
415     getbits(in, 0);  
416     getbits(ou, 1);  
417     getbits(ex, 2);  
418 #undef getbits  
419     if (uap->tv) {  
420         error = copyin((caddr_t) uap->tv, (caddr_t) & atv,  
421                         sizeof(atv));  
422         if (error)  
423             goto done;  
424         if (itimerfix(&atv)) {  
425             error = EINVAL;  
426             goto done;  
427         }  
428         s = splclock();  
429         timevaladd(&atv, (struct timeval *) &time);  
430         timo = hzto(&atv);  
431         /*  
432          * Avoid inadvertently sleeping forever.  
433          */  
434         if (timo == 0)  
435             timo = 1;  
436         splx(s);  
437     } else  
438         timo = 0;  
-----sys_generic.c-----
```

## Validation and setup

390-410

Two arrays of three descriptor sets are allocated. They are cleared by bzero. The first argument, maximum number of descriptors associated with fd\_set, is set to the sum of the number of descriptors currently allocated to each fd\_set and the number of descriptors to be added with nd bits (1 bit for each descriptor). For example, if the maximum number of descriptors is 256 (FD\_SETSIZE), fd\_set is represented by 32 integers (NFDBITS), and nd is 65, then:

$$ni = \text{howmany}(65, 32) \times 4 = 3 \times 4 = 12$$

where howmany (x,y) returns the number of y-bit integers needed to represent x.

## Copy file descriptor sets from process

411-418

The getbits macro uses copyin to transfer the file descriptor sets from the process to the three descriptor sets in ibits. If a descriptor is copied, its value is copied from the process.

## Setup timeout value

419-438

If tv is null, timo is set to 0 and select will wait until the hardware clock by itimerfix. The current time is timevaladd. The number of clock ticks until the timeout is saved in timo. If the resulting timeout is 0, select returns from blocking and implements the nonblocking structure.

The second half of select, shown in Figure 16.5 indicated by the process and returns when one timer expires, or a signal occurs.

**Figure 16.54. select function**

```

439    retry:
440        ncoll = nselcoll;
441        p->p_flag |= P_SELECT;
442        error = selscan(p, ibits, obits, uap->nd, retval);
443        if (error || *retval)
444            goto done;
445        s = splhigh();
446        /* this should be timercmp(&time, &atv, >=) */
447        if (uap->tv && (time.tv_sec > atv.tv_sec ||
448             time.tv_sec == atv.tv_sec && time.tv_usec >= atv.tv_usec)) {
449            splx(s);
450            goto done;
451        }
452        if ((p->p_flag & P_SELECT) == 0 || nselcoll != ncoll) {
453            splx(s);
454            goto retry;
455        }
456        p->p_flag &= ~P_SELECT;
457        error = tsleep((caddr_t) & selwait, PSOCK | PCATCH, "select", timo);
458        splx(s);
459        if (error == 0)
460            goto retry;
461    done:
462        p->p_flag &= ~P_SELECT;
463        /* select is not restarted after signals... */
464        if (error == ERESTART)
465            error = EINTR;
466        if (error == EWOULDBLOCK)
467            error = 0;
468 #define putbits(name, x) \
469     if (uap->name && \
470         (error2 = copyout((caddr_t)&obits[x], (caddr_t)uap->name, ni))) \
471         error = error2;
472     if (error == 0) {
473         int      error2;
474         putbits(in, 0);
475         putbits(ou, 1);
476         putbits(ex, 2);
477 #undef putbits
478     }
479     return (error);
480 }

```

sys\_generic.c

## Scan file descriptors

439-442

The loop that starts at retry continues until selection of the global integer nselcoll is saved and the P process's control block. If either of these change while checking the file descriptors, it indicates that they changed because of interrupt processing and selection.

selscan looks at every descriptor set in the three descriptor sets and adds the matching descriptor in the output set if the

**Figure 16.55. selscan**

```
481 selscan(p, ibits, obits, nfd, retval)                                sys_generic.c
482 struct proc *p;
483 fd_set *ibits, *obits;
484 int     nfd, *retval;
485 {
486     struct filedesc *fdp = p->p_fd;
487     int      msk, i, j, fd;
488     fd_mask bits;
489     struct file *fp;
490     int      n = 0;
491     static int flag[3] =
492     {FREAD, FWRITE, 0};
493     for (msk = 0; msk < 3; msk++) {
494         for (i = 0; i < nfd; i += NFDBITS) {
495             bits = ibits[msk].fds_bits[i / NFDBITS];
496             while ((j = ffs(bits)) && (fd = i + --j) < nfd) {
497                 bits &= ~(1 << j);
498                 fp = fdp->fd_ofiles[fd];
499                 if (fp == NULL)
500                     return (EBADF);
501                 if ((*fp->f_ops->fo_select) (fp, flag[msk], p)) {
502                     FD_SET(fd, &obits[msk]);
503                     n++;
504                 }
505             }
506         }
507     }
508     *retval = n;
509     return (0);
510 }
```

Error or some descriptors are ready

443-444

Return immediately if an error occurred or if a descriptor is ready.

Timeout expired?

**445-451**

If the process supplied a time limit and the current value is less than the timeout value, return immediately.

## **Status changed during selscan**

**452-455**

selscan can be interrupted by protocol processes. If the interrupt occurs during the interrupt, P\_SELECT and nselcoll are the descriptors.

## **Wait for buffer changes**

**456-460**

All processes calling select use selwait as the waiting function. With [Figure 16.60](#) we show that this call can be shared by more than one process. If a process receives an error, select jumps to retry to rescan the descriptors.

## **Ready to return**

**461-480**

At done, P\_SELECT is cleared, ERESTART is changed to 0, and EWOULDBLOCK is changed to 0. These changes occur when a signal occurs during select and 0 is returned.

The output descriptor sets are copied back to the descriptor sets.

## selscan Function

The heart of select is the selscan function shown. It takes a descriptor set in one of the three descriptor sets, selscan, associated with the bit and dispatches control to the descriptor. For sockets, this

## Locate descriptors to be monitored

481-496

The first for loop iterates through each of the three descriptor sets and exception. The second for loop interates within each descriptor set. The inner while loop is executed once for every 32 bits (NFDBITS).

The inner while loop checks all the descriptors in the descriptor set extracted from the current descriptor set and sets the position of the first 1 bit. For example, if bits is 1000 (with 28 leading 0s), ff000000000000000000000000000000, it will return 0x00000000000000000000000000000001.

## Poll descriptor

497-500

From i and the return value of ffs, the descriptor is computed and stored in fd. The bit is cleared in descriptor set), the file structure associated with fo\_select is called.

The second argument to fo\_select is one of the is the index of the outer for loop. So the first time it is FREAD, the second time it is FWRI. EBADF is returned if the descriptor is not valid.

## Descriptor is ready

501-504

When a descriptor is found to be ready, the main descriptor set and n (the number of matches) is

505-510

The loops continue until all the descriptors are descriptors is returned in \*retval.

## soo\_select Function

For every descriptor that selscan finds in the in function referenced by the fo\_select pointer in [15.5](#)) associated with the descriptor. In this text socket descriptors and the soo\_select function :

**Figure 16.56. soo\_select**

```
sys_socket.c
105 soo_select(fp, which, p)
106 struct file *fp;
107 int      which;
108 struct proc *p;
109 {
110     struct socket *so = (struct socket *) fp->f_data;
111     int      s = splnet();
112     switch (which) {
113         case FREAD:
114             if (soreadable(so)) {
115                 splx(s);
116                 return (1);
117             }
118             selrecord(p, &so->so_rcv.sb_sel);
119             so->so_rcv.sb_flags |= SB_SEL;
120             break;
121         case FWRITE:
122             if (sowriteable(so)) {
123                 splx(s);
124                 return (1);
125             }
126             selrecord(p, &so->so_snd.sb_sel);
127             so->so_snd.sb_flags |= SB_SEL;
128             break;
129         case 0:
130             if (so->so_oobmark || (so->so_state & SS_RCVATMARK)) {
131                 splx(s);
132                 return (1);
133             }
134             selrecord(p, &so->so_rcv.sb_sel);
135             so->so_rcv.sb_flags |= SB_SEL;
136             break;
137     }
138     splx(s);
139     return (0);
140 }
```

sys\_socket.c

105-112

Each time soo\_select is called, it checks the sta

descriptor is ready relative to the conditions sp returns 1 immediately. If the descriptor is not r socket's receive or send buffer to indicate that buffer and then soo\_select returns 0.

Figure 16.52 showed the read, write, and except macros soreadable and sownable. Here we see that the macros soreadable and sownable are defined in sys/socket.h.

## Is socket readable?

113-120

The soreadable macro is:

```
#define soreadable(so) \
    ((so)->so_rcv.sb_cc >= (so)->so_lowat) || \
    ((so)->so_state & SS_CANTRCVR) || \
    (so)->so_qlen || (so)->so_error
```

Since the receive low-water mark for UDP and TCP is zero, the socket is readable if any data is in the receive buffer. If the connection is closed, if any connections are ready to be read, or if there is an error pending.

## Is socket writeable?

121-128

The sowriteable macro is:

```
#define sowriteable(so) \
    (sbspace(&(so)->so_snd) >= 1 \
     & ((so)->so_state&SS_ISCONNEC \
        & ((so)->so_proto->pr_flags& \
        ((so)->so_state & SS_CANTSEN \
        (so)->so_error))
```

The default send low-water mark for UDP and TCP is always true because sbspace is always equal to sb\_lowat, and a connection is always established.

For TCP, the socket is not writeable when the free space is less than 2048 bytes. The other cases are described in the code.

**Are there any exceptional conditions pending?**

129-140

For exceptions, so\_oobmark and the SS\_RCVAL exception exists until the process has a mark in the data stream.

## selrecord Function

Figure 16.57 shows the definition of the selinfo and receive buffer (the sb\_sel member from Figure 16.56).

**Figure 16.57. selinfo structure**

```
41 struct selinfo {
42     pid_t    si_pid;           /* process to be notified */
43     short    si_flags;        /* 0 or SI_COLL */
44 };
```

41-44

When only one process has called select for a given descriptor, si\_pid contains the process ID of the waiting process. When additional processes have called select for the same descriptor, SI\_COLL is set in si\_flags. This is the only flag currently defined for si\_flags.

The selrecord function shown in Figure 16.58 is called when a process has selected a descriptor that is not ready. The function records the process ID of the process that is awakened by the protocol processing.

**Figure 16.58. selrecord**

```
522 void  
523 selrecord(selector, sip)  
524 struct proc *selector;  
525 struct selinfo *sip;  
526 {  
527     struct proc *p;  
528     pid_t mypid;  
529     mypid = selector->p_pid;  
530     if (sip->si_pid == mypid)  
531         return;  
532     if (sip->si_pid && (p = pfind(sip->si_pid)) &&  
533         p->p_wchan == (caddr_t) & selwait)  
534         sip->si_flags |= SI_COLL;  
535     else  
536         sip->si_pid = mypid;  
537 }
```

sys\_generic.c

## Already selecting on this descriptor

522-531

The first argument to selrecord points to the process. The second argument points to the sel (so\_snd.sb\_sel or so\_rcv.sb\_sel). If this process's selinfo record for this socket buffer, the function example, the process called select with the same descriptor.

## Select collision with another process?

532-534

If another process is already selecting on this b

## No collision

535-537

If there is no other process already selecting or of the current process is saved in si\_pid.

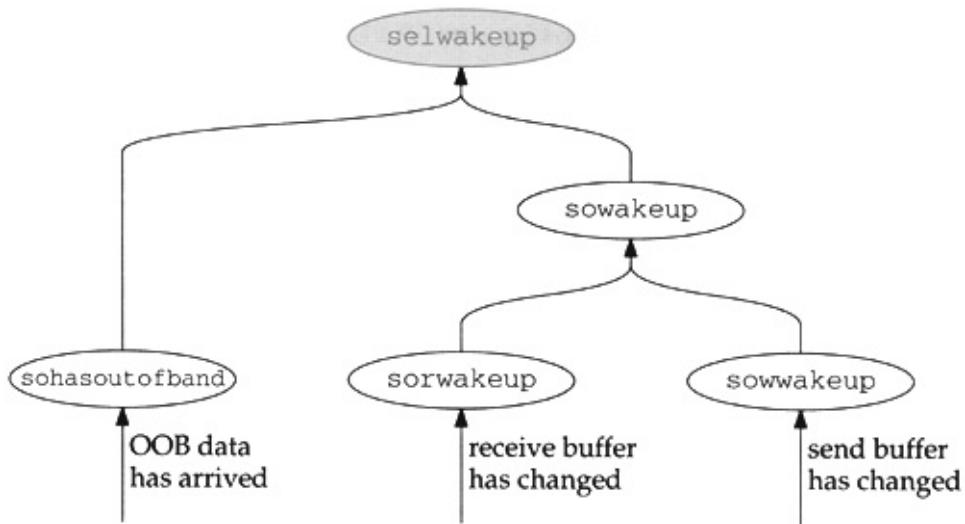
## selwakeup Function

When protocol processing changes the state of process is selecting on the buffer, Net/3 can im the run queue based on the information it finds

When the state changes and there is more than buffer (SI\_COLL is set), Net/3 has no way of determining if the process is still interested in the buffer. When we discussed the select function, we pointed out that every process that calls select is still interested in the buffer when calling tsleep. This means the corresponding processes that are blocked in select even those that are blocked on the buffer.

Figure 16.59 shows how selwakeup is called.

## Figure 16.59. selwakeup



The protocol processing layer is responsible for calling one of the functions listed at the bottom occurs that changes the state of a socket. The bottom of [Figure 16.59](#) cause selwakeup to be on the socket to be scheduled to run.

selwakeup is shown in [Figure 16.60](#).

**Figure 16.60. selwakeup**

```

541 void
542 selwakeup(sip)
543 struct selinfo *sip;
544 {
545     struct proc *p;
546     int     s;
547
548     if (sip->si_pid == 0)
549         return;
550     if (sip->si_flags & SI_COLL) {
551         nselcoll++;
552         sip->si_flags &= ~SI_COLL;
553         wakeup((caddr_t) & selwait);
554     }
555     p = pfind(sip->si_pid);
556     sip->si_pid = 0;
557     if (p != NULL) {
558         s = splhigh();
559         if (p->p_wchan == (caddr_t) & selwait) {
560             if (p->p_stat == SSLEEP)
561                 setrunnable(p);
562             else
563                 unsleep(p);
564         } else if (p->p_flag & P_SELECT)
565             p->p_flag &= ~P_SELECT;
566         splx(s);
567     }
568 }
```

sys\_generic.c

## 541-548

If si\_pid is 0, there is no process selecting on tl returns immediately.

## Wake all processes during a collision

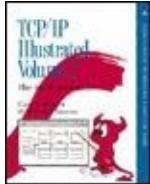
## 549-553

If more than one process is selecting on the aff incremented, the collision flag is cleared, and e awakened. As mentioned with [Figure 16.54](#), ns descriptors if the buffers change before the pro ([Exercise 16.9](#)).

If the process identified by si\_pid is waiting on the process is waiting on some other wait chan  
The process can be waiting on some other wait a valid descriptor and then selscan finds a bad descriptor sets. selscan returns EBADF, but the record is not reset. Later, when selwakeup runs process identified by sel\_pid is no longer waitin selinfo information is ignored.

Only one process is awakened during selwakeup sharing the same descriptor (i.e., the same soc the machines to which the authors had access, confirms the statement that select collisions are

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 16. Socket I/O

### 16.14 Summary

In this chapter we looked at the read, write, and select system calls for sockets.

We saw that `sosend` handles all output between the socket layer and the protocol processing layer and that `soreceive` handles all input.

The organization of the send buffer and receive buffers was described, as well as the default values and semantics of the high-water and low-water marks for the buffers.

The last part of the chapter discussed the implementation of select. We showed that when only one process is selecting on a

descriptor, the protocol processing layer will awaken only the process identified in the selinfo structure. When there is a collision and more than one process is selecting on a descriptor, the protocol layer has no choice but to awaken every process that is selecting on *any* descriptor.

## Exercises

What happens to resid in sosend when an unsigned integer larger **16.1** than the maximum positive signed integer is passed in the write system call?

When sosend puts less than MCLBYTES of data in a cluster, space is reduced by the full **16.2** MCLBYTES and may become negative, which terminates the loop that fills mbufs for atomic protocols. Is this a problem?

Datagram and stream protocols have very different semantics.

Divide the `sosend` and `soreceive` functions each into two functions, one to handle messages, and one to handle streams. Other than making the code clearer, what are the advantages of making this change?

For `PR_ATOMIC` protocols, each write call specifies an implicit message boundary. The socket layer delivers the message as a single unit to the protocol. The `MSG_EOR` flag allows a process to specify explicit message boundaries. Why is the implicit technique insufficient?

What happens when `sosend` cannot immediately acquire a lock on the send buffer when the socket descriptor is marked as nonblocking and the process does not specify `MSG_DONTWAIT`?

Under what circumstances would  
sb\_cc < sb\_hiwat yet sbspace  
**16.6** would report no free space? Why  
should a process be blocked in this  
case?

Why isn't the length of a control  
**16.7** message copied back to the process  
by recvit as is the name length?

**16.8** Why does soreceive clear  
MSG\_EOR?

What might happen if the nselcoll  
**16.9** code were removed from select and  
selwakeup?

Modify the select system call to  
**16.10** return the time remaining in the  
timer when select returns.

---

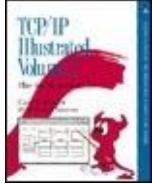
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 17. Socket Options

[Section 17.1. Introduction](#)

[Section 17.2. Code Introduction](#)

[Section 17.3. setsockopt System Call](#)

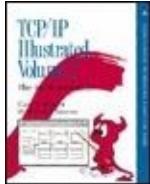
[Section 17.4. getsockopt System Call](#)

[Section 17.5. fcntl and ioctl System Calls](#)

[Section 17.6. getsockname System Call](#)

[Section 17.7. getpeername System Call](#)

[Section 17.8. Summary](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 17. Socket Options

### 17.1 Introduction

We complete our discussion of the socket layer in this chapter by discussing several system calls that modify the behavior of sockets.

The `setsockopt` and `getsockopt` system calls were introduced in [Section 8.8](#), where we described the options that provide access to IP features. In this chapter we show the implementation of these two system calls and the socket-level options that are controlled through them.

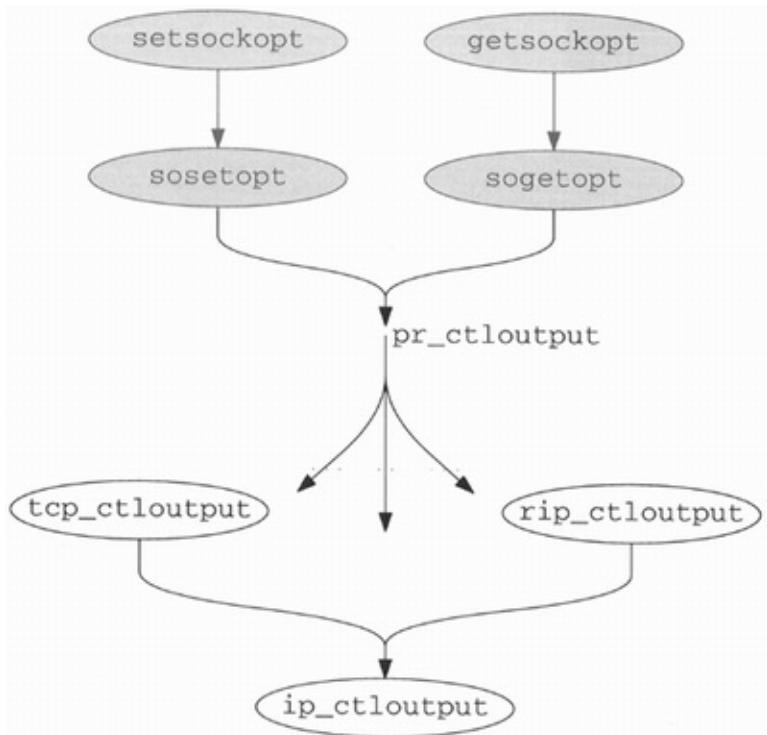
The `ioctl` function was introduced in [Section 4.4](#), where we described the protocol-independent `ioctl` commands for

network interface configuration. In [Section 6.7](#) we described the IP specific ioctl commands used to assign network masks as well as unicast, broadcast, and destination addresses. In this chapter we describe the implementation of ioctl and the related features of the fcntl function.

Finally, we describe the getsockname and getpeername system calls, which return address information for sockets and connections.

[Figure 17.1](#) shows the functions that implement the socket option system calls. The shaded functions are described in this chapter.

**Figure 17.1. setsockopt and getsockopt system calls.**



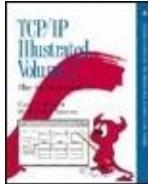
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 17. Socket Options

## 17.2 Code Introduction

The code in this chapter comes from the four files listed in [Figure 17.2](#).

**Figure 17.2. Files discussed in this chapter.**

File	Description
kern/kern_descrip.c	fcntl system call
kern/uipc_syscalls.c	setsockopt, getsockopt, getsockname, and getpeername system calls
kern/uipc_socket.c	socket layer processing for setsockopt and getsockopt
kern/sys_socket.c	ioctl system call for sockets

## Global Variables and Statistics

No new global variables are introduced and no statistics are collected by the system

calls we describe in this chapter.

---

**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)

## Chapter 17. Socket Options

### 17.3 setsockopt System Call

Figure 8.29 listed the different protocol levels to use with the getsockopt(2) and setsockopt(2) system calls (see Figure 17.3). In this chapter we focus on the SC level.

**Figure 17.3. setsockopt System Call**

option name	option type	Variable	Description
<i>SO_SNDBUF</i>	int	<i>so_snd.sb_hiwat</i>	send buffer high-water mark
<i>SO_RCVBUF</i>	int	<i>so_rcv.sb_hiwat</i>	receive buffer high-water mark
<i>SO SNDLOWAT</i>	int	<i>so_snd.sb_lowat</i>	send buffer low-water mark
<i>SO_RCVLOWAT</i>	int	<i>so_rcv.sb_lowat</i>	receive buffer low-water mark
<i>SO_SNDTIMEO</i>	struct timeval	<i>so_snd.sb_timeo</i>	send timeout
<i>SO_RCVTIMEO</i>	struct timeval	<i>so_rcv.sb_timeo</i>	receive timeout
<i>SO_DEBUG</i>	int	<i>so_options</i>	record debugging information for this socket
<i>SO_REUSEADDR</i>	int	<i>so_options</i>	socket can reuse a local address
<i>SO_REUSEPORT</i>	int	<i>so_options</i>	socket can reuse a local port
<i>SO_KEEPALIVE</i>	int	<i>so_options</i>	protocol probes idle connections
<i>SO_DONTROUTE</i>	int	<i>so_options</i>	bypass routing tables
<i>SO_BROADCAST</i>	int	<i>so_options</i>	socket allows broadcast messages
<i>SO_USELOOPBACK</i>	int	<i>so_options</i>	routing domain sockets only; sending messages
			process receives its own routing messages
<i>SO_OOBINLINE</i>	int	<i>so_options</i>	protocol queues out-of-band data inline
<i>SO_LINGER</i>	struct linger	<i>so_linger</i>	socket lingers on close
<i>SO_ERROR</i>	int	<i>so_error</i>	get error status and clear; getsockopt only
<i>SO_TYPE</i>	int	<i>so_type</i>	get socket type; getsockopt only
other			ENOPROTOOPT returned

The prototype for setsockopt is

```
int setsockopt(int s, int level,
```

Figure 17.4 shows the code for this system call

Figure 17.4. setsockopt

```
565 struct setsockopt_args {
566     int      s;
567     int      level;
568     int      name;
569     caddr_t  val;
570     int      valsiz;
571 };
572 setsockopt(p, uap, retval)
573 struct proc *p;
574 struct setsockopt_args *uap;
575 int    *retval;
576 {
577     struct file *fp;
578     struct mbuf *m = NULL;
579     int      error;
580     if (error = getsock(p->p_fd, uap->s, &fp))
581         return (error);
582     if (uap->valsiz > MLEN)
583         return (EINVAL);
584     if (uap->val) {
585         m = m_get(M_WAIT, MT_SOOPTS);
586         if (m == NULL)
587             return (ENOBUFS);
588         if (error = copyin(uap->val, mtod(m, caddr_t),
589                           (u_int) uap->valsiz)) {
590             (void) m_free(m);
591             return (error);
592         }
593         m->m_len = uap->valsiz;
594     }
595     return (so_setopt((struct socket *) fp->f_data, uap->level,
596                       uap->name, m));
597 }
```

uipc\_syscalls.c

565-597

getsock locates the file structure for the socket

copied from the process into an mbuf allocated  
be no more than MLEN bytes in length, so if va  
soisetopt is called and its value is returned.

## soisetopt Function

This function processes all the socket-level opti  
function for the protocol associated with the so

Figure 17.5.

```
752 soisetopt(so, level, optname, m0)                                uipc_socket.c
753 struct socket *so;
754 int      level, optname;
755 struct mbuf *m0;
756 {
757     int      error = 0;
758     struct mbuf *m = m0;
759
760     if (level != SOL_SOCKET) {
761         if (so->so_proto && so->so_proto->pr_ctloutput)
762             return ((*so->so_proto->pr_ctloutput)
763                      (PRCO_SETOPT, so, level, optname, &m0));
764         error = ENOPROTOOPT;
765     } else {
766         switch (optname) {
767
768             /* socket option processing */
769
770             default:
771                 error = ENOPROTOOPT;
772                 break;
773             }
774             if (error == 0 && so->so_proto && so->so_proto->pr_ctloutput) {
775                 (void) ((*so->so_proto->pr_ctloutput)
776                          (PRCO_SETOPT, so, level, optname, &m0));
777                 m = NULL;           /* freed by protocol */
778             }
779         }
780     }
781     bad:
782     if (m)
783         (void) m_free(m);
784     return (error);
785 }
```

752-764

If the option is not for the socket level (SOL\_SOCKET), it is passed to the underlying protocol. Note that the protocol's pr\_usrreq function. [Figure 17.6](#) shows which functions handle each option.

**Figure 17.6. pr\_usrreq Functions**

Protocol	pr_ctloutput Function	Reference
UDP	ip_ctloutput	Section 8.8
TCP	tcp_ctloutput	Section 30.6
ICMP		
IGMP	rip_ctloutput and ip_ctloutput	Section 8.8 and Section 32.8
raw IP		

765

The switch statement handles the socket-level options.

841-844

An unrecognized option causes ENOPROTOOPT and the error is released.

845-855

Unless an error occurs, control always falls through to the associated protocol in case the protocol layer needs to handle the option. None of the Internet protocols expect to receive options from the user.

Notice that the return value from the call to the option is not expected by the protocol. m is protocol layer is responsible for releasing the m

Figure 17.7 shows the linger option and the opti

## Figure 17.7. sosetopt for

```
766     case SO_LINGER:
767         if (m == NULL || m->m_len != sizeof(struct linger)) {
768             error = EINVAL;
769             goto bad;
770         }
771         so->so_linger = mtod(m, struct linger *)->l_linger;
772         /* fall thru... */
773     case SO_DEBUG:
774     case SO_KEEPALIVE:
775     case SO_DONTROUTE:
776     case SO_USELOOPBACK:
777     case SO_BROADCAST:
778     case SO_REUSEADDR:
779     case SO_REUSEPORT:
780     case SO_OOBINLINE:
781         if (m == NULL || m->m_len < sizeof(int)) {
782             error = EINVAL;
783             goto bad;
784         }
785         if (*mtod(m, int *))
786             so->so_options |= optname;
787         else
788             so->so_options &= ~optname;
789         break;
```

766-772

The linger option expects the process to pass a

```
struct linger {
    int      l_onoff;      /* option */
    int      l_linger;     /* linger */
```

```
};
```

After making sure the process has passed data member is copied into `so_linger`. The option is `SO_LINGER`. `SO_LINGER` was described in [Section 7.7.3-7.8.9](#)

[7.7.3-7.8.9](#)

These options are boolean flags set when the program is passed. The first check makes sure an integer is passed, then sets or clears the appropriate option.

[Figure 17.8](#) shows the socket buffer options.

**Figure 17.8. `soSetopt` function**

```

790     case SO_SNDBUF:
791     case SO_RCVBUF:
792     case SO SNDLOWAT:
793     case SO RCVLOWAT:
794         if (m == NULL || m->m_len < sizeof(int)) {
795             error = EINVAL;
796             goto bad;
797         }
798     switch (optname) {
799         case SO_SNDBUF:
800         case SO_RCVBUF:
801             if (sbreserve(optname == SO_SNDBUF ?
802                         &so->so_snd : &so->so_rcv,
803                         (u_long) * mtod(m, int *)) == 0) {
804                 error = ENOBUFS;
805                 goto bad;
806             }
807             break;
808         case SO SNDLOWAT:
809             so->so_snd.sb_lowat = *mtod(m, int *);
810             break;
811         case SO RCVLOWAT:
812             so->so_rcv.sb_lowat = *mtod(m, int *);
813             break;
814         }
815     break;

```

uipc\_socket.c

## 790-815

This set of options changes the size of the send buffer. If the required integer has been provided for sbreserve, it adjusts the high-water mark but does not change it. For SO\_RCVLOWAT, the low-water marks are adjusted.

Figure 17.9 shows the timeout options.

**Figure 17.9. `sosetopt1`**

---

```

816     case SO_SNDFTIMEO:
817     case SO_RCVTIMEO:
818     {
819         struct timeval *tv;
820         short val;
821
822         if (m == NULL || m->m_len < sizeof(*tv)) {
823             error = EINVAL;
824             goto bad;
825         }
826         tv = mtod(m, struct timeval *);
827         if (tv->tv_sec > SHRT_MAX / hz - hz) {
828             error = EDOM;
829             goto bad;
830         }
831         val = tv->tv_sec * hz + tv->tv_usec / tick;
832
833         switch (optname) {
834
835             case SO_SNDFTIMEO:
836                 so->so_snd.sb_timeo = val;
837                 break;
838             case SO_RCVTIMEO:
839                 so->so_rcv.sb_timeo = val;
840                 break;
841         }
842     }

```

---

*uipc\_socket.c*

816-824

The timeout value for SO\_SNDFTIMEO and SO\_RCVTIMEO is stored in the `sb_timeo` field of the `SO_F` structure. If the right amount of data is not available in the socket buffer, the function returns -1.

825-830

The time interval stored in the `timeval` structure is represented as clock ticks, it fits within a short (`short`) integer type.

The code on line 826 is incorrect. The time interval

$$tv\_sec \times hz + \frac{tv\_usec}{tick} > SHRT\_MAX$$

where

$$\text{tick} = \frac{1,000,000}{\text{hz}} \text{ and } \text{SHRT\_MAX} = 32767$$

So EDOM should be returned if

$$\text{tv\_sec} > \frac{\text{SHRT\_MAX}}{\text{hz}} - \frac{\text{tv\_usec}}{\text{tick} \times \text{hz}} = \frac{\text{SHRT\_MAX}}{\text{hz}} - \frac{\text{tv\_usec}}{1,000,000}$$

The last term in this equation is not hz as speci

if (tv->tv\_sec\*hz + tv->tv\_usec/

but see Exercise 17.3 for more discussion.

831-840

The converted time, val, is saved in the send or amount of time a process will wait for data in the Sections 16.7 and 16.12 for details.

The timeout values are passed as the last argument to the process is limited to 65535 ticks. At 100 Hz, t

## Chapter 17. Socket Options

### 17.4 getsockopt System Call

getsockopt returns socket and protocol options

```
int getsockopt(int s, int level,
```

The code is shown in [Figure 17.10](#).

**Figure 17.10.** getsockopt

---

```

598 struct getsockopt_args {
599     int      s;
600     int      level;
601     int      name;
602     caddr_t val;
603     int      *avalsize;
604 };
605 getsockopt(p, uap, retval)
606 struct proc *p;
607 struct getsockopt_args *uap;
608 int    *retval;
609 {
610     struct file *fp;
611     struct mbuf *m = NULL;
612     int      valsiz, error;
613     if (error = getsock(p->p_fd, uap->s, &fp))
614         return (error);
615     if (uap->val) {
616         if (error = copyin((caddr_t) uap->avalsize, (caddr_t) & valsiz,
617                           sizeof(valsiz)))
618             return (error);
619     } else
620         valsiz = 0;
621     if ((error = so getopt((struct socket *) fp->f_data, uap->level,
622                           uap->name, &m)) == 0 && uap->val && valsiz && m != NULL) {
623         if (valsiz > m->m_len)
624             valsiz = m->m_len;
625         error = copyout(mtod(m, caddr_t), uap->val, (u_int) valsiz);
626         if (error == 0)
627             error = copyout((caddr_t) & valsiz,
628                             (caddr_t) uap->avalsize, sizeof(valsiz));
629     }
630     if (m != NULL)
631         (void) m_free(m);
632     return (error);
633 }

```

---

— *uipc\_syscalls.c*

## 598-633

The code should look pretty familiar by now. The user buffer is copied into the kernel, and so getopt is called. The data returned by so getopt is copied out to the user buffer. If the user buffer is too small, a new length of the buffer. It is possible that the user buffer is too small to hold the data. In that case, the function returns -1.

## so getopt Function

As with soisetopt, the sogetopt function handles options to the protocol associated with the socket shown in Figure 17.11.

Figure 17.11. sogetopt

```
856 sogetopt(so, level, optname, mp)                                uipc_socket.c
857 struct socket *so;
858 int      level, optname;
859 struct mbuf **mp;
860 {
861     struct mbuf *m;
862
863     if (level != SOL_SOCKET) {
864         if (so->so_proto && so->so_proto->pr_ctloutput) {
865             return ((*so->so_proto->pr_ctloutput)
866                      (PRCO_GETOPT, so, level, optname, mp));
867         } else
868             return (ENOPROTOOPT);
869     } else {
870         m = m_get(M_WAIT, MT_SOOPTS);
871         m->m_len = sizeof(int);
872
873         switch (optname) {
874
875             /* socket option processing */
876
877         default:
878             (void) m_free(m);
879             return (ENOPROTOOPT);
880         }
881         *mp = m;
882         return (0);
883     }
884 }
```

856-871

As with soisetopt, options that do not pertain to protocol level through the PRCO\_GETOPT proto option in the mbuf pointed to by \*mp.

For socket-level options, a standard mbuf is allo-

an integer, so m\_len is set to the size of an integer by the code in the switch statement.

918-925

If the default case is taken by the switch, the n Otherwise, after the switch statement, the pointer function returns, getsockopt copies the option from mbuf.

In [Figure 17.12](#) the linger option and the options processed.

**Figure 17.12. sogetopt function:**

```
872     case SO_LINGER:
873         m->m_len = sizeof(struct linger);
874         mtod(m, struct linger *)->l_onoff =
875             so->so_options & SO_LINGER;
876         mtod(m, struct linger *)->l_linger = so->so_linger;
877         break;
878
879     case SO_USELOOPBACK:
880     case SO_DONTROUTE:
881     case SO_DEBUG:
882     case SO_KEEPALIVE:
883     case SO_REUSEADDR:
884     case SO_REUSEPORT:
885     case SO_BROADCAST:
886     case SO_OOBINLINE:
887         *mtod(m, int *) = so->so_options & optname;
888         break;
```

872-877

The SO\_LINGER option requires two copies, one linger time into l\_linger.

878-887

The remaining options are implemented as booleans which results in a nonzero value if the option is set. The return value is not necessarily 1 when the flag is set.

In the next part of so getopt (Figure 17.13), the code handles the following options:

**Figure 17.13. so getopt function**

```
888     case SO_TYPE:
889         *mtod(m, int *) = so->so_type;
890         break;
891
892     case SO_ERROR:
893         *mtod(m, int *) = so->so_error;
894         so->so_error = 0;
895         break;
896
897     case SO_SNDBUF:
898         *mtod(m, int *) = so->so_snd.sb_hiwat;
899         break;
900
901     case SO_RCVBUF:
902         *mtod(m, int *) = so->so_rcv.sb_hiwat;
903         break;
904
905     case SO_SNDLOWAT:
906         *mtod(m, int *) = so->so_snd.sb_lowat;
907         break;
908
909     case SO_RCVLOWAT:
910         *mtod(m, int *) = so->so_rcv.sb_lowat;
911         break;
```

888-906

Each option is copied as an integer into the memory buffer. In the kernel (e.g., the high-water and low-water marks), the `so_error` is cleared once the value is copied into. The `getsockopt` changes the state of the socket.

The fourth and last part of so getopt is shown if SO\_RCVTIMEO options are handled.

Figure 17.14. so getopt

```
907     case SO_SNDTIMEO:  
908     case SO_RCVTIMEO:  
909         {  
910             int      val = (optname == SO_SNDTIMEO ?  
911                           so->so_snd.sb_timeo : so->so_rcv.sb_timeo);  
912  
913             m->m_len = sizeof(struct timeval);  
914             mtod(m, struct timeval *)->tv_sec = val / hz;  
915             mtod(m, struct timeval *)->tv_usec =  
916                           (val % hz) / tick;  
917             break;  
918         }  
919     }  
920  
921     if (m->m_len <= 0)  
922         m->m_len = 1;  
923     else if (m->m_len > 1)  
924         m->m_len = 2;  
925     else if (m->m_len > 2)  
926         m->m_len = 3;  
927     else if (m->m_len > 3)  
928         m->m_len = 4;  
929     else if (m->m_len > 4)  
930         m->m_len = 5;  
931     else if (m->m_len > 5)  
932         m->m_len = 6;  
933     else if (m->m_len > 6)  
934         m->m_len = 7;  
935     else if (m->m_len > 7)  
936         m->m_len = 8;  
937     else if (m->m_len > 8)  
938         m->m_len = 9;  
939     else if (m->m_len > 9)  
940         m->m_len = 10;  
941     else if (m->m_len > 10)  
942         m->m_len = 11;  
943     else if (m->m_len > 11)  
944         m->m_len = 12;  
945     else if (m->m_len > 12)  
946         m->m_len = 13;  
947     else if (m->m_len > 13)  
948         m->m_len = 14;  
949     else if (m->m_len > 14)  
950         m->m_len = 15;  
951     else if (m->m_len > 15)  
952         m->m_len = 16;  
953     else if (m->m_len > 16)  
954         m->m_len = 17;  
955     else if (m->m_len > 17)  
956         m->m_len = 18;  
957     else if (m->m_len > 18)  
958         m->m_len = 19;  
959     else if (m->m_len > 19)  
960         m->m_len = 20;  
961     else if (m->m_len > 20)  
962         m->m_len = 21;  
963     else if (m->m_len > 21)  
964         m->m_len = 22;  
965     else if (m->m_len > 22)  
966         m->m_len = 23;  
967     else if (m->m_len > 23)  
968         m->m_len = 24;  
969     else if (m->m_len > 24)  
970         m->m_len = 25;  
971     else if (m->m_len > 25)  
972         m->m_len = 26;  
973     else if (m->m_len > 26)  
974         m->m_len = 27;  
975     else if (m->m_len > 27)  
976         m->m_len = 28;  
977     else if (m->m_len > 28)  
978         m->m_len = 29;  
979     else if (m->m_len > 29)  
980         m->m_len = 30;  
981     else if (m->m_len > 30)  
982         m->m_len = 31;  
983     else if (m->m_len > 31)  
984         m->m_len = 32;  
985     else if (m->m_len > 32)  
986         m->m_len = 33;  
987     else if (m->m_len > 33)  
988         m->m_len = 34;  
989     else if (m->m_len > 34)  
990         m->m_len = 35;  
991     else if (m->m_len > 35)  
992         m->m_len = 36;  
993     else if (m->m_len > 36)  
994         m->m_len = 37;  
995     else if (m->m_len > 37)  
996         m->m_len = 38;  
997     else if (m->m_len > 38)  
998         m->m_len = 39;  
999     else if (m->m_len > 39)  
1000        m->m_len = 40;  
1001  
1002     if (m->m_len > 40)  
1003         m->m_len = 41;  
1004  
1005     if (m->m_len > 41)  
1006         m->m_len = 42;  
1007  
1008     if (m->m_len > 42)  
1009         m->m_len = 43;  
1010  
1011     if (m->m_len > 43)  
1012         m->m_len = 44;  
1013  
1014     if (m->m_len > 44)  
1015         m->m_len = 45;  
1016  
1017     if (m->m_len > 45)  
1018         m->m_len = 46;  
1019  
1020     if (m->m_len > 46)  
1021         m->m_len = 47;  
1022  
1023     if (m->m_len > 47)  
1024         m->m_len = 48;  
1025  
1026     if (m->m_len > 48)  
1027         m->m_len = 49;  
1028  
1029     if (m->m_len > 49)  
1030         m->m_len = 50;  
1031  
1032     if (m->m_len > 50)  
1033         m->m_len = 51;  
1034  
1035     if (m->m_len > 51)  
1036         m->m_len = 52;  
1037  
1038     if (m->m_len > 52)  
1039         m->m_len = 53;  
1040  
1041     if (m->m_len > 53)  
1042         m->m_len = 54;  
1043  
1044     if (m->m_len > 54)  
1045         m->m_len = 55;  
1046  
1047     if (m->m_len > 55)  
1048         m->m_len = 56;  
1049  
1050     if (m->m_len > 56)  
1051         m->m_len = 57;  
1052  
1053     if (m->m_len > 57)  
1054         m->m_len = 58;  
1055  
1056     if (m->m_len > 58)  
1057         m->m_len = 59;  
1058  
1059     if (m->m_len > 59)  
1060         m->m_len = 60;  
1061  
1062     if (m->m_len > 60)  
1063         m->m_len = 61;  
1064  
1065     if (m->m_len > 61)  
1066         m->m_len = 62;  
1067  
1068     if (m->m_len > 62)  
1069         m->m_len = 63;  
1070  
1071     if (m->m_len > 63)  
1072         m->m_len = 64;  
1073  
1074     if (m->m_len > 64)  
1075         m->m_len = 65;  
1076  
1077     if (m->m_len > 65)  
1078         m->m_len = 66;  
1079  
1080     if (m->m_len > 66)  
1081         m->m_len = 67;  
1082  
1083     if (m->m_len > 67)  
1084         m->m_len = 68;  
1085  
1086     if (m->m_len > 68)  
1087         m->m_len = 69;  
1088  
1089     if (m->m_len > 69)  
1090         m->m_len = 70;  
1091  
1092     if (m->m_len > 70)  
1093         m->m_len = 71;  
1094  
1095     if (m->m_len > 71)  
1096         m->m_len = 72;  
1097  
1098     if (m->m_len > 72)  
1099         m->m_len = 73;  
1100  
1101     if (m->m_len > 73)  
1102         m->m_len = 74;  
1103  
1104     if (m->m_len > 74)  
1105         m->m_len = 75;  
1106  
1107     if (m->m_len > 75)  
1108         m->m_len = 76;  
1109  
1110     if (m->m_len > 76)  
1111         m->m_len = 77;  
1112  
1113     if (m->m_len > 77)  
1114         m->m_len = 78;  
1115  
1116     if (m->m_len > 78)  
1117         m->m_len = 79;  
1118  
1119     if (m->m_len > 79)  
1120         m->m_len = 80;  
1121  
1122     if (m->m_len > 80)  
1123         m->m_len = 81;  
1124  
1125     if (m->m_len > 81)  
1126         m->m_len = 82;  
1127  
1128     if (m->m_len > 82)  
1129         m->m_len = 83;  
1130  
1131     if (m->m_len > 83)  
1132         m->m_len = 84;  
1133  
1134     if (m->m_len > 84)  
1135         m->m_len = 85;  
1136  
1137     if (m->m_len > 85)  
1138         m->m_len = 86;  
1139  
1140     if (m->m_len > 86)  
1141         m->m_len = 87;  
1142  
1143     if (m->m_len > 87)  
1144         m->m_len = 88;  
1145  
1146     if (m->m_len > 88)  
1147         m->m_len = 89;  
1148  
1149     if (m->m_len > 89)  
1150         m->m_len = 90;  
1151  
1152     if (m->m_len > 90)  
1153         m->m_len = 91;  
1154  
1155     if (m->m_len > 91)  
1156         m->m_len = 92;  
1157  
1158     if (m->m_len > 92)  
1159         m->m_len = 93;  
1160  
1161     if (m->m_len > 93)  
1162         m->m_len = 94;  
1163  
1164     if (m->m_len > 94)  
1165         m->m_len = 95;  
1166  
1167     if (m->m_len > 95)  
1168         m->m_len = 96;  
1169  
1170     if (m->m_len > 96)  
1171         m->m_len = 97;  
1172  
1173     if (m->m_len > 97)  
1174         m->m_len = 98;  
1175  
1176     if (m->m_len > 98)  
1177         m->m_len = 99;  
1178  
1179     if (m->m_len > 99)  
1180         m->m_len = 100;  
1181  
1182     if (m->m_len > 100)  
1183         m->m_len = 101;  
1184  
1185     if (m->m_len > 101)  
1186         m->m_len = 102;  
1187  
1188     if (m->m_len > 102)  
1189         m->m_len = 103;  
1190  
1191     if (m->m_len > 103)  
1192         m->m_len = 104;  
1193  
1194     if (m->m_len > 104)  
1195         m->m_len = 105;  
1196  
1197     if (m->m_len > 105)  
1198         m->m_len = 106;  
1199  
1200     if (m->m_len > 106)  
1201         m->m_len = 107;  
1202  
1203     if (m->m_len > 107)  
1204         m->m_len = 108;  
1205  
1206     if (m->m_len > 108)  
1207         m->m_len = 109;  
1208  
1209     if (m->m_len > 109)  
1210         m->m_len = 110;  
1211  
1212     if (m->m_len > 110)  
1213         m->m_len = 111;  
1214  
1215     if (m->m_len > 111)  
1216         m->m_len = 112;  
1217  
1218     if (m->m_len > 112)  
1219         m->m_len = 113;  
1220  
1221     if (m->m_len > 113)  
1222         m->m_len = 114;  
1223  
1224     if (m->m_len > 114)  
1225         m->m_len = 115;  
1226  
1227     if (m->m_len > 115)  
1228         m->m_len = 116;  
1229  
1230     if (m->m_len > 116)  
1231         m->m_len = 117;  
1232  
1233     if (m->m_len > 117)  
1234         m->m_len = 118;  
1235  
1236     if (m->m_len > 118)  
1237         m->m_len = 119;  
1238  
1239     if (m->m_len > 119)  
1240         m->m_len = 120;  
1241  
1242     if (m->m_len > 120)  
1243         m->m_len = 121;  
1244  
1245     if (m->m_len > 121)  
1246         m->m_len = 122;  
1247  
1248     if (m->m_len > 122)  
1249         m->m_len = 123;  
1250  
1251     if (m->m_len > 123)  
1252         m->m_len = 124;  
1253  
1254     if (m->m_len > 124)  
1255         m->m_len = 125;  
1256  
1257     if (m->m_len > 125)  
1258         m->m_len = 126;  
1259  
1260     if (m->m_len > 126)  
1261         m->m_len = 127;  
1262  
1263     if (m->m_len > 127)  
1264         m->m_len = 128;  
1265  
1266     if (m->m_len > 128)  
1267         m->m_len = 129;  
1268  
1269     if (m->m_len > 129)  
1270         m->m_len = 130;  
1271  
1272     if (m->m_len > 130)  
1273         m->m_len = 131;  
1274  
1275     if (m->m_len > 131)  
1276         m->m_len = 132;  
1277  
1278     if (m->m_len > 132)  
1279         m->m_len = 133;  
1280  
1281     if (m->m_len > 133)  
1282         m->m_len = 134;  
1283  
1284     if (m->m_len > 134)  
1285         m->m_len = 135;  
1286  
1287     if (m->m_len > 135)  
1288         m->m_len = 136;  
1289  
1290     if (m->m_len > 136)  
1291         m->m_len = 137;  
1292  
1293     if (m->m_len > 137)  
1294         m->m_len = 138;  
1295  
1296     if (m->m_len > 138)  
1297         m->m_len = 139;  
1298  
1299     if (m->m_len > 139)  
1300         m->m_len = 140;  
1301  
1302     if (m->m_len > 140)  
1303         m->m_len = 141;  
1304  
1305     if (m->m_len > 141)  
1306         m->m_len = 142;  
1307  
1308     if (m->m_len > 142)  
1309         m->m_len = 143;  
1310  
1311     if (m->m_len > 143)  
1312         m->m_len = 144;  
1313  
1314     if (m->m_len > 144)  
1315         m->m_len = 145;  
1316  
1317     if (m->m_len > 145)  
1318         m->m_len = 146;  
1319  
1320     if (m->m_len > 146)  
1321         m->m_len = 147;  
1322  
1323     if (m->m_len > 147)  
1324         m->m_len = 148;  
1325  
1326     if (m->m_len > 148)  
1327         m->m_len = 149;  
1328  
1329     if (m->m_len > 149)  
1330         m->m_len = 150;  
1331  
1332     if (m->m_len > 150)  
1333         m->m_len = 151;  
1334  
1335     if (m->m_len > 151)  
1336         m->m_len = 152;  
1337  
1338     if (m->m_len > 152)  
1339         m->m_len = 153;  
1340  
1341     if (m->m_len > 153)  
1342         m->m_len = 154;  
1343  
1344     if (m->m_len > 154)  
1345         m->m_len = 155;  
1346  
1347     if (m->m_len > 155)  
1348         m->m_len = 156;  
1349  
1350     if (m->m_len > 156)  
1351         m->m_len = 157;  
1352  
1353     if (m->m_len > 157)  
1354         m->m_len = 158;  
1355  
1356     if (m->m_len > 158)  
1357         m->m_len = 159;  
1358  
1359     if (m->m_len > 159)  
1360         m->m_len = 160;  
1361  
1362     if (m->m_len > 160)  
1363         m->m_len = 161;  
1364  
1365     if (m->m_len > 161)  
1366         m->m_len = 162;  
1367  
1368     if (m->m_len > 162)  
1369         m->m_len = 163;  
1370  
1371     if (m->m_len > 163)  
1372         m->m_len = 164;  
1373  
1374     if (m->m_len > 164)  
1375         m->m_len = 165;  
1376  
1377     if (m->m_len > 165)  
1378         m->m_len = 166;  
1379  
1380     if (m->m_len > 166)  
1381         m->m_len = 167;  
1382  
1383     if (m->m_len > 167)  
1384         m->m_len = 168;  
1385  
1386     if (m->m_len > 168)  
1387         m->m_len = 169;  
1388  
1389     if (m->m_len > 169)  
1390         m->m_len = 170;  
1391  
1392     if (m->m_len > 170)  
1393         m->m_len = 171;  
1394  
1395     if (m->m_len > 171)  
1396         m->m_len = 172;  
1397  
1398     if (m->m_len > 172)  
1399         m->m_len = 173;  
1400  
1401     if (m->m_len > 173)  
1402         m->m_len = 174;  
1403  
1404     if (m->m_len > 174)  
1405         m->m_len = 175;  
1406  
1407     if (m->m_len > 175)  
1408         m->m_len = 176;  
1409  
1410     if (m->m_len > 176)  
1411         m->m_len = 177;  
1412  
1413     if (m->m_len > 177)  
1414         m->m_len = 178;  
1415  
1416     if (m->m_len > 178)  
1417         m->m_len = 179;  
1418  
1419     if (m->m_len > 179)  
1420         m->m_len = 180;  
1421  
1422     if (m->m_len > 180)  
1423         m->m_len = 181;  
1424  
1425     if (m->m_len > 181)  
1426         m->m_len = 182;  
1427  
1428     if (m->m_len > 182)  
1429         m->m_len = 183;  
1430  
1431     if (m->m_len > 183)  
1432         m->m_len = 184;  
1433  
1434     if (m->m_len > 184)  
1435         m->m_len = 185;  
1436  
1437     if (m->m_len > 185)  
1438         m->m_len = 186;  
1439  
1440     if (m->m_len > 186)  
1441         m->m_len = 187;  
1442  
1443     if (m->m_len > 187)  
1444         m->m_len = 188;  
1445  
1446     if (m->m_len > 188)  
1447         m->m_len = 189;  
1448  
1449     if (m->m_len > 189)  
1450         m->m_len = 190;  
1451  
1452     if (m->m_len > 190)  
1453         m->m_len = 191;  
1454  
1455     if (m->m_len > 191)  
1456         m->m_len = 192;  
1457  
1458     if (m->m_len > 192)  
1459         m->m_len = 193;  
1460  
1461     if (m->m_len > 193)  
1462         m->m_len = 194;  
1463  
1464     if (m->m_len > 194)  
1465         m->m_len = 195;  
1466  
1467     if (m->m_len > 195)  
1468         m->m_len = 196;  
1469  
1470     if (m->m_len > 196)  
1471         m->m_len = 197;  
1472  
1473     if (m->m_len > 197)  
1474         m->m_len = 198;  
1475  
1476     if (m->m_len > 198)  
1477         m->m_len = 199;  
1478  
1479     if (m->m_len > 199)  
1480         m->m_len = 200;  
1481  
1482     if (m->m_len > 200)  
1483         m->m_len = 201;  
1484  
1485     if (m->m_len > 201)  
1486         m->m_len = 202;  
1487  
1488     if (m->m_len > 202)  
1489         m->m_len = 203;  
1490  
1491     if (m->m_len > 203)  
1492         m->m_len = 204;  
1493  
1494     if (m->m_len > 204)  
1495         m->m_len = 205;  
1496  
1497     if (m->m_len > 205)  
1498         m->m_len = 206;  
1499  
1500     if (m->m_len > 206)  
1501         m->m_len = 207;  
1502  
1503     if (m->m_len > 207)  
1504         m->m_len = 208;  
1505  
1506     if (m->m_len > 208)  
1507         m->m_len = 209;  
1508  
1509     if (m->m_len > 209)  
1510         m->m_len = 210;  
1511  
1512     if (m->m_len > 210)  
1513         m->m_len = 211;  
1514  
1515     if (m->m_len > 211)  
1516         m->m_len = 212;  
1517  
1518     if (m->m_len > 212)  
1519         m->m_len = 213;  
1520  
1521     if (m->m_len > 213)  
1522         m->m_len = 214;  
1523  
1524     if (m->m_len > 214)  
1525         m->m_len = 215;  
1526  
1527     if (m->m_len > 215)  
1528         m->m_len = 216;  
1529  
1530     if (m->m_len > 216)  
1531         m->m_len = 217;  
1532  
1533     if (m->m_len > 217)  
1534         m->m_len = 218;  
1535  
1536     if (m->m_len > 218)  
1537         m->m_len = 219;  
1538  
1539     if (m->m_len > 219)  
1540         m->m_len = 220;  
1541  
1542     if (m->m_len > 220)  
1543         m->m_len = 221;  
1544  
1545     if (m->m_len > 221)  
1546         m->m_len = 222;  
1547  
1548     if (m->m_len > 222)  
1549         m->m_len = 223;  
1550  
1551     if (m->m_len > 223)  
1552         m->m_len = 224;  
1553  
1554     if (m->m_len > 224)  
1555         m->m_len = 225;  
1556  
1557     if (m->m_len > 225)  
1558         m->m_len = 226;  
1559  
1560     if (m->m_len > 226)  
1561         m->m_len = 227;  
1562  
1563     if (m->m_len > 227)  
1564         m->m_len = 228;  
1565  
1566     if (m->m_len > 228)  
1567         m->m_len = 229;  
1568  
1569     if (m->m_len > 229)  
1570         m->m_len = 230;  
1571  
1572     if (m->m_len > 230)  
1573         m->m_len = 231;  
1574  
1575     if (m->m_len > 231)  
1576         m->m_len = 232;  
1577  
1578     if (m->m_len > 232)  
1579         m->m_len = 233;  
1580  
1581     if (m->m_len > 233)  
1582         m->m_len = 234;  
1583  
1584     if (m->m_len > 234)  
1585         m->m_len = 235;  
1586  
1587     if (m->m_len > 235)  
1588         m->m_len = 236;  
1589  
1590     if (m->m_len > 236)  
1591         m->m_len = 237;  
1592  
1593     if (m->m_len > 237)  
1594         m->m_len = 238;  
1595  
1596     if (m->m_len > 238)  
1597         m->m_len = 239;  
1598  
1599     if (m->m_len > 239)  
1600         m->m_len = 240;  
1601  
1602     if (m->m_len > 240)  
1603         m->m_len = 241;  
1604  
1605     if (m->m_len > 241)  
1606         m->m_len = 242;  
1607  
1608     if (m->m_len > 242)  
1609         m->m_len = 243;  
1610  
1611     if (m->m_len > 243)  
1612         m->m_len = 244;  
1613  
1614     if (m->m_len > 244)  
1615         m->m_len = 245;  
1616  
1617     if (m->m_len > 245)  
1618         m->m_len = 246;  
1619  
1620     if (m->m_len > 246)  
1621         m->m_len = 247;  
1622  
1623     if (m->m_len > 247)  
1624         m->m_len = 248;  
1625  
1626     if (m->m_len > 248)  
1627         m->m_len = 249;  
1628  
1629     if (m->m_len > 249)  
1630         m->m_len = 250;  
1631  
1632     if (m->m_len > 250)  
1633         m->m_len = 251;  
1634  
1635     if (m->m_len > 251)  
1
```

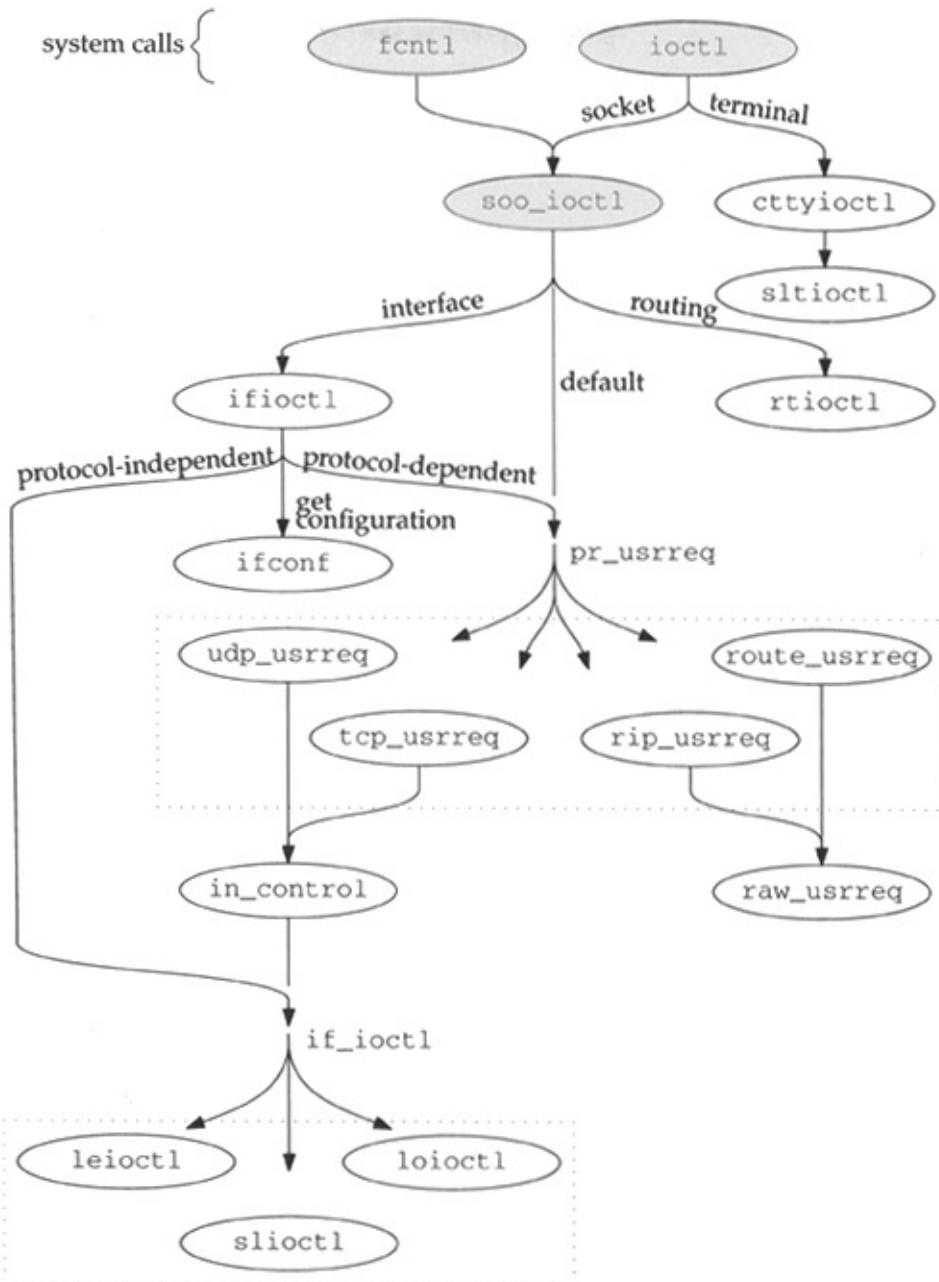
## Chapter 17. Socket Options

### 17.5 fcntl and ioctl System Calls

Due more to history than intent, several features can only be accessed from either ioctl or fcntl. We have described the ioctl commands and have mentioned fcntl s

Figure 17.15 highlights the functions described

**Figure 17.15. fcntl and ioctl functions**



The prototypes for `ioctl` and `fcntl` are:

```
int ioctl(int fd, unsigned long
```

```
int fcntl(int fd, int cmd, ... /
```

**Figure 17.16** summarizes the features of these relate to sockets. We show the traditional const since they appear in the code. For Posix compa be used instead of FNONBLOCK, and O\_ASYNC FASYNC.

**Figure 17.16. fcntl and ioctl co**

Description	fcntl	ioctl
enable or disable nonblocking semantics by turning <code>SS_NBIO</code> on or off in <code>so_state</code>	FNONBLOCK file status flag	FIONBIO command
enable or disable asynchronous notification by turning <code>SB_ASYNC</code> on or off in <code>sb_flags</code>	FASYNC file status flag	FIOASYNC command
set or get <code>so_pgid</code> , which is the target process or process group for SIGIO and SIGURG signals	<code>F_SETOWN</code> or <code>F_GETOWN</code>	SIOCSPGRP or SIOCGPGRP commands
get number of bytes in receive buffer; return <code>so_rcv.sb_cc</code>		FIONREAD
return OOB synchronization mark; the <code>SS_RCVATMARK</code> flag in <code>so_state</code>		SIOCATMARK

## fcntlCode

**Figure 17.17** shows an overview of the fcntl fur

**Figure 17.17. fcntl system call**

```

133 struct fcntl_args {
134     int      fd;
135     int      cmd;
136     int      arg;
137 };
138 /* ARGSUSED */
139 fcntl(p, uap, retval)
140 struct proc *p;
141 struct fcntl_args *uap;
142 int      *retval;
143 {
144     struct filedesc *fdp = p->p_fd;
145     struct file *fp;
146     struct vnode *vp;
147     int      i, tmp, error, flg = F_POSIX;
148     struct flock fl;
149     u_int    newmin;
150     if ((unsigned) uap->fd >= fdp->fd_nfiles || 
151         (fp = fdp->fd_ofiles[uap->fd]) == NULL)
152         return (EBADF);
153     switch (uap->cmd) {
154         /* command processing */
155     default:
156         return (EINVAL);
157     }
158     /* NOTREACHED */
159 }
```

— *kern\_descrip.c*

## 133-153

After verifying that the descriptor refers to an open file, the code processes the requested command.

## 253-257

If the command is not recognized, fcntl returns EINVAL.

Figure 17.18 shows only the cases from fcntl that handle commands.

**Figure 17.18. fcntl system call: source code**

```

168     case F_GETFL:
169         *retval = OFLAGS(fp->f_flag);
170         return (0);
171     case F_SETFL:
172         fp->f_flag &= ~FCNTLFLAGS;
173         fp->f_flag |= FFLAGS(uap->arg) & FCNTLFLAGS;
174         tmp = fp->f_flag & FNONBLOCK;
175         error = (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
176         if (error)
177             return (error);
178         tmp = fp->f_flag & FASYNC;
179         error = (*fp->f_ops->fo_ioctl) (fp, FIOASYNC, (caddr_t) & tmp, p);
180         if (!error)
181             return (0);
182         fp->f_flag &= ~FNONBLOCK;
183         tmp = 0;
184         (void) (*fp->f_ops->fo_ioctl) (fp, FIONBIO, (caddr_t) & tmp, p);
185         return (error);
186     case F_GETOWN:
187         if (fp->f_type == DTTYPE_SOCKET) {
188             *retval = ((struct socket *) fp->f_data)->so_pgid;
189             return (0);
190         }
191         error = (*fp->f_ops->fo_ioctl)
192             (fp, (int) TIOCGRPGRP, (caddr_t) retval, p);
193         *retval = -*retval;
194         return (error);
195     case F_SETOWN:
196         if (fp->f_type == DTTYPE_SOCKET) {
197             ((struct socket *) fp->f_data)->so_pgid = uap->arg;
198             return (0);
199         }
200         if (uap->arg <= 0) {
201             uap->arg = -uap->arg;
202         } else {
203             struct proc *pl = pfind(uap->arg);
204             if (pl == 0)
205                 return (ESRCH);
206             uap->arg = pl->p_pgrp->pg_id;
207         }
208         return ((*fp->f_ops->fo_ioctl)
209             (fp, (int) TIOCSPGRP, (caddr_t) & uap->arg, p));

```

*kern\_descrip.c*

## 168-185

F\_GETFL returns the current file status flags as descriptor and F\_SETFL sets the flags. The new and FASYNC are passed to the associated socket for sockets is the so\_ioctl function described previously. The third call to fo\_ioctl is made only if the second call set the FNONBLOCK flag, but should instead restore the original setting.

186-209

F\_GETOWN returns so\_pgid, the process or pro with the socket. For a descriptor other than a s ioctl command is passed to the associated fo\_ic assigns a new value to so\_pgid.

For a descriptor other than a socket, the proces function, but for sockets, the value is checked j in sohasoutofband and in sowakeup.

## ioctl Code

We skip the ioctl system call itself and start wit 17.20, since most of the code in ioctl duplicates with Figure 17.17. We've already shown that th commands to rtioctl, interface commands to ific commands to the pr\_usrreq function of the unc

55-68

A few commands are handled by soo\_ioctl direc nonblocking semantics if \*data is nonzero, and As we have seen, this flag affects the accept,cc calls as well as the various read and write syste

69-79

FIOASYNC enables or disables asynchronous I/O. If there is activity on a socket, sowakeup gets called, the SIGIO signal is sent to the process or process group.

80-88

FIONREAD returns the number of bytes available for reading. SIOCSPGRP sets the process group associated with a socket. SIOCGPGRP gets it. so\_pgid is used as a target for the process group we just described and for the SIGURG signal which arrives for a socket. The signal is sent when the so\_hasoutofband function.

89-92

SIOCATMARK returns true if the socket is at the synchronization mark, false otherwise.

ioctl commands, the FIOxxx and SIOxxx constants have the structure illustrated in Figure 17.19.

**Figure 17.19. The structure of an ioctl command.**



## Figure 17.20. `soo_ioctl` function

```
55 soo_ioctl(fp, cmd, data, p)                                sys_socket.c
56 struct file *fp;
57 int      cmd;
58 caddr_t data;
59 struct proc *p;
60 {
61     struct socket *so = (struct socket *) fp->f_data;
62     switch (cmd) {
63     case FIONBIO:
64         if (*(int *) data)
65             so->so_state |= SS_NBIO;
66         else
67             so->so_state &= ~SS_NBIO;
68         return (0);
69     case FIOASYNC:
70         if (*(int *) data) {
71             so->so_state |= SS_ASYNC;
72             so->so_rcv.sb_flags |= SB_ASYNC;
73             so->so_snd.sb_flags |= SB_ASYNC;
74         } else {
75             so->so_state &= ~SS_ASYNC;
76             so->so_rcv.sb_flags &= ~SB_ASYNC;
77             so->so_snd.sb_flags &= ~SB_ASYNC;
78         }
79         return (0);
80     case FIONREAD:
81         *(int *) data = so->so_rcv.sb_cc;
82         return (0);
83     case SIOCSPGRP:
84         so->so_pgid = *(int *) data;
85         return (0);
86     case SIOCGPGRP:
87         *(int *) data = so->so_pgid;
88         return (0);
89     case SIOCATMARK:
90         *(int *) data = (so->so_state & SS_RCVATMARK) != 0;
91         return (0);
92     }
93     /*
94      * Interface/routing/protocol specific ioctls:
95      * interface and routing ioctls should have a
96      * different entry since a socket's unnecessary
97      */
98     if (IOCGROUP(cmd) == 'i')
99         .return (ifioctl(so, cmd, data, p));
100    if (IOCGROUP(cmd) == 'r')
101        return (rtioctl(cmd, data, p));
102    return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
103                                         (struct mbuf *) cmd, (struct mbuf *) data, (struct mbuf *) 0));
104 }
```

sys\_socket.c

If the third argument to `ioctl` is used as input, *input* is set. If the fourth argument is used as output, *output* is set. If the fifth argument is set, *length* is the size of the argument in bytes.

commands are in the same *group* but each command is within the group. The macros in [Figure 17.21](#) extract information from an ioctl command.

**Figure 17.21. ioctl command**

Macro	Description
IOCPARM_LEN (cmd)	the <i>length</i> from <i>cmd</i>
IOCBASECMD (cmd)	the command with <i>length</i> set to 0
IOCGROUP (cmd)	the <i>group</i> from <i>cmd</i>

93-104

The macro IOCGROUP extracts the 8-bit *group* from an ioctl command. Interface commands are handled by ifioctl. Router commands are processed by rtioctl. All other commands are passed to the protocol through the PRU\_CONTROL request.

As we describe in [Chapter 19](#), Net/2 introduced routing tables in which messages are passed between protocols through a socket created in the PF\_ROUTE domain. This replaces the ioctl method shown here. rtioctl returns ENOTSUPP in kernels that do not have compatibility support.

## Chapter 17. Socket Options

### 17.6 getsockname System Call

The prototype for this system call is:

```
int getsockname(int fd, caddr_t
```

getsockname retrieves the local address bound places it in the buffer pointed to by *asa*. This is kernel has selected an address during an implicit process specified a wildcard address ([Section 2](#) explicit call to bind. The getsockname system call is described in [Section 17.22](#).

**Figure 17.22. getsockname system call**

---

```

682 struct getsockname_args {
683     int      fdes;
684     caddr_t asa;
685     int      *alen;
686 };
687 getsockname(p, uap, retval)
688 struct proc *p;
689 struct getsockname_args *uap;
690 int      *retval;
691 {
692     struct file *fp;
693     struct socket *so;
694     struct mbuf *m;
695     int      len, error;
696     if (error = getsock(p->p_fd, uap->fdes, &fp))
697         return (error);
698     if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
699         return (error);
700     so = (struct socket *) fp->f_data;
701     m = m_getclr(M_WAIT, MT_SONAME);
702     if (m == NULL)
703         return (ENOBUFS);
704     if (error = (*so->so_proto->pr_usrreq) (so, PRU_SOCKETADDR, 0, m, 0))
705         goto bad;
706     if (len > m->m_len)
707         len = m->m_len;
708     error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len);
709     if (error == 0)
710         error = copyout((caddr_t) & len, (caddr_t) uap->alen,
711                         sizeof(len));
712     bad:
713     m_freem(m);
714     return (error);
715 }

```

---

uipc\_syscalls.c

## 682-715

getsock locates the file structure for the descriptor. The buffer specified by the process is copied from the file structure. This is the first call to m\_getclr that we've seen. It allocates a standard mbuf and clears it with bzero. The protocol layer is responsible for returning the local address. A PRU\_SOCKETADDR request is issued.

If the address is larger than the buffer specified, it is silently truncated when it is copied out to the parameter. The length is updated to the number of bytes copied out to the parameter.

the mbuf is released and getsockname returns.

---

**Team-Fly**

[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Hartman  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 17. Socket Options

### 17.7 getpeername System Call

The prototype for this system call is:

```
int getpeername(int fd, caddr_t
```

The `getpeername` system call returns the address of the connection associated with the specified socket descriptor. It is often called when a server is invoked through a process that calls `accept` (i.e., any server socket). A server doesn't have access to the peer address and must use `getpeername`. The returned address can be checked against an access list for the application, and the connection closed if the address is not on the list.

Some protocols, such as TP4, utilize this function to determine if an incoming connection should be rejected or confirmed. A connection associated with a socket returned by `accept` is not complete and must be confirmed before the connection is accepted.

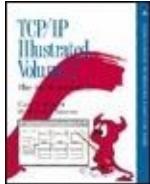
Based on the address returned by getpeername, close the connection or implicitly confirm the connection or receiving data. This feature is irrelevant for TCP, as it doesn't make a connection available to accept until the handshake is complete. [Figure 17.23](#) shows the code for the function.

**Figure 17.23. getpeername system call**

```
719 struct getpeername_args {
720     int      fdes;
721     caddr_t  asa;
722     int      *alen;
723 };
724 getpeername(p, uap, retval)
725 struct proc *p;
726 struct getpeername_args *uap;
727 int      *retval;
728 {
729     struct file *fp;
730     struct socket *so;
731     struct mbuf *m;
732     int      len, error;
733     if (error = getsock(p->p_fd, uap->fdes, &fp))
734         return (error);
735     so = (struct socket *) fp->f_data;
736     if ((so->so_state & (SS_ISCONNECTED | SS_ISCONFIRMING)) == 0)
737         return (ENOTCONN);
738     if (error = copyin((caddr_t) uap->alen, (caddr_t) & len, sizeof(len)))
739         return (error);
740     m = m_getclr(M_WAIT, MT_SONAME);
741     if (m == NULL)
742         return (ENOBUFS);
743     if (error = (*so->so_proto->pr_usrreq) (so, PRU_PEERADDR, 0, m, 0))
744         goto bad;
745     if (len > m->m_len)
746         len = m->m_len;
747     if (error = copyout(mtod(m, caddr_t), (caddr_t) uap->asa, (u_int) len))
748         goto bad;
749     error = copyout((caddr_t) & len, (caddr_t) uap->alen, sizeof(len));
750 bad:
751     m_freem(m);
752     return (error);
753 }
```

The code here is almost identical to the getsockname function. It first locates the socket and ENOTCONN is returned if the socket is not yet connected to a peer or if the connection is in a disconnected state (e.g., TP4). If it is connected, the size of the buffer is determined from the process and an mbuf is allocated to hold the data. A PRU\_PEERADDR request is issued to get the remote address from the protocol layer. The address and the length of the address are copied from the kernel mbuf to the buffer in the user space and the mbuf is released and the function returns.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 17. Socket Options

### 17.8 Summary

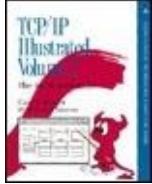
In this chapter we discussed the six functions that modify the semantics of a socket. Socket options are processed by `setsockopt` and `getsockopt`. Additional options, some of which are not unique to sockets, are handled by `fcntl` and `ioctl`. Finally, connection information is available through `getsockname` and `getpeername`.

### Exercises

Why do you think options are limited  
**17.1** to the size of a standard mbuf  
(`MHLEN`, 128 bytes)?

Why does the code at the end of  
**17.2** [Figure 17.7](#) work for the SO\_LINGER option?

There is a problem with the suggested code used to test the timeval structure in [Figure 17.9](#) since **17.3** `tv->tv_sec * hz` may cause an overflow. Suggest a change to the code to solve this problem.



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 18. Radix Tree Routing Tables

[Section 18.1. Introduction](#)

[Section 18.2. Routing Table Structure](#)

[Section 18.3. Routing Sockets](#)

[Section 18.4. Code Introduction](#)

[Section 18.5. Radix Node Data Structures](#)

[Section 18.6. Routing Structures](#)

[Section 18.7. Initialization: route\\_init and rtable\\_init Functions](#)

[Section 18.8. Initialization: rn\\_init and rn\\_inithead Functions](#)

[Section 18.9. Duplicate Keys and Mask Lists](#)

[Section 18.10. rn\\_match Function](#)

## Section 18.11. rn\_search Function

## Section 18.12. Summary

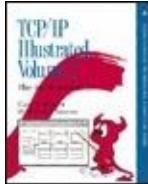
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.1 Introduction

The routing performed by IP, when it searches the routing table and decides which interface to send a packet out on, is a *routing mechanism*. This differs from a *routing policy*, which is a set of rules that decides which routes go into the routing table. The Net/3 kernel implements the routing mechanism while a routing daemon, typically routed or gated, implements the routing policy. The structure of the routing table must recognize that the packet forwarding occurs frequently hundreds or thousands of times a second on a busy system while routing policy changes are less frequent.

Routing is a detailed issue and we divide our discussion into three chapters.

- This chapter looks at the structure of the radix tree routing tables used by the Net/3 packet forwarding code. The tables are consulted by IP every time a packet is sent (since IP must determine which local interface receives the packet) and every time a packet is forwarded.
- [Chapter 19](#) looks at the functions that interface between the kernel and the radix tree functions, and also at the routing messages that are exchanged between the kernel and routing processes normally the routing daemons that implement the routing policy. These messages allow a process to modify the kernel's routing table (add a route, delete a route, etc.) and let the kernel notify the daemons when an asynchronous event occurs that might affect the routing policy (a redirect is received, an interface goes down, and so on).

- Chapter 20 presents the routing sockets that are used to exchange routing messages between the kernel and a process.
- 

Team-Fly



[◀ Previous](#)

[Next ▶](#)

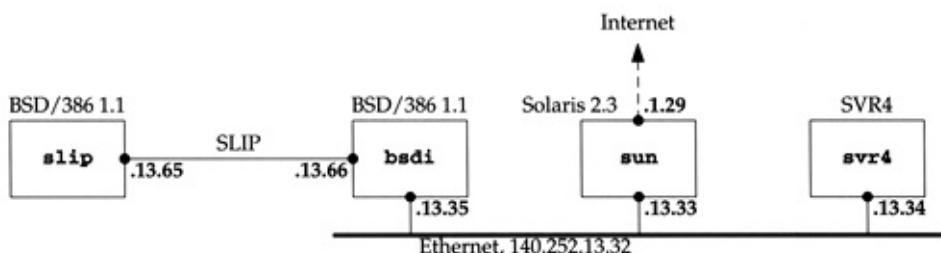
[Top](#)

## Chapter 18. Radix Tree Routing Tables

### 18.2 Routing Table Structure

Before looking at the internal structure of the routing table, we need to understand the type of information contained in the table. [Figure 18.1](#) is the bottom part of [Figure 1.17](#): the four systems on the author's Ethernet.

**Figure 18.1. Subnet used for routing table**



[Figure 18.2](#) shows the routing table for bsdi in

## Figure 18.2. Routing table on the host

```
bsdi $ netstat -rn
Routing tables

Internet:
Destination      Gateway          Flags     Refs      Use   Interface
default          140.252.13.33   UG S        0       3    le0
127              127.0.0.1      UG S R      0       2    lo0
127.0.0.1        127.0.0.1      U H         1      55    lo0
128.32.33.5      140.252.13.33  UGHS       2       16    le0
140.252.13.32    link#1         U C         0       0    le0
140.252.13.33    8:0:20:3:f6:42  U H L       11     55146  le0
140.252.13.34    0:0:c0:c2:9b:26  U H L       0       3    le0
140.252.13.35    0:0:c0:6f:2d:40  U H L       1      12    lo0
140.252.13.65    140.252.13.66  U H         0      41    s10
224              link#1         U C         0       0    le0
224.0.0.1         link#1         U H L       0       5    le0
```

We have modified the "Flags" column from the output, making it easier to see which flags are various entries.

The routes in this table were entered as follows: 5, 8, and 9 are performed at system initialization; /etc/netstart shell script is executed.

### 1. A default route is added by the route command, pointing to the host sun (140.252.13.33), which connects to the Internet.

- The entry for network 127 is typically created by a daemon such as gated, or it can be entered with the route command in the /etc/netstart file. This entry causes packets sent to this network, other than referred to host 127.0.0.1 (which are covered by the more specific route to 127.0.0.1).

entered in the next step), to be rejected by the driver ([Figure 5.27](#)).

- The entry for the loopback interface (127.0.0.1) is configured by ifconfig.
- The entry for vangogh.cs.berkeley.edu (128.3) is created by hand using the route command. It specifies the same router as the default route (140.252.13.3). This is a host-specific route, instead of using the default route for this host, which allows routing metrics to be stored in the route entry. These metrics can optionally be set by the administrator using the metric option. They are used by TCP each time a connection is established to a host on the destination network. The metrics are updated by TCP when a connection is closed. We describe these metrics with [Figure 27.3](#).
- The interface 1e0 is initialized using the ifconfig command. This causes the entry for network 140.252.13.0 (140.252.13.1) to be entered into the routing table.
- The entries for the other two hosts on the Ethernet (140.252.13.33) and svr4 (140.252.13.34), were learned via ARP, as we describe in [Chapter 21](#). These are temporary entries that are removed if they are not used for a long period of time.
- The entry for the local host, 140.252.13.35, is

first time the host's own IP address is referenced, the interface is the loopback, meaning any IP datagrams with the host's own IP address are looped back instead. The automatic creation of this entry is new with 4.4 and will be described in [Section 21.13](#).

- The entry for the host 140.252.13.65 is created because the SLIP interface is configured by ifconfig.
- The route command adds the route to network 140.252.13.0 via the Ethernet interface.
- The entry for the multicast group 224.0.0.1 (the broadcast group) was created by running the Ping program to that address 224.0.0.1. This is also a temporary entry that is removed if not used for a certain period of time.

The "Flags" column in [Figure 18.2](#) needs a brief explanation. [Figure 18.25](#) provides a list of all the possible flags.

U The route is up.

G The route is to a gateway (router). This is called a *dynamic route*. If this flag is not set, the destination is directly connected; this is called a *direct route*.

H The route is to a host, that is, the destination host address. If this flag is *not* set, the route network, and the destination is a network address, network ID, or a combination of a network ID and a host ID. The netstat command doesn't show it, but the network route also contains a network mask. This route has an implied mask of all one bits.

S The route is static. The three entries created by the route command in [Figure 18.2](#) are static.

C The route is cloned to create new routes. Two entries in this routing table have this flag set: (1) the route to the local Ethernet (140.252.13.32), which is cloned to create the host-specific routes of other hosts on the local Ethernet, and (2) the route for multicast groups, which is cloned to create specific multicast group routes such as 224.0.0.1.

L The route contains a link-layer address. The link layer address is the MAC address that ARP clones from the Ethernet network route. This flag is set if the link flag is set. This applies to unicast and multicast addresses.

R The loopback driver (the normal interface for this flag) rejects all datagrams that use this r

The ability to enter a route with the "reject provided in Net/2. It provides a simple way preventing datagrams destined to network appearing outside the host. See also [Exerc](#)

Before 4.3BSD Reno, two distinct routing tables were maintained by the addresses: one for host routes and one for network routes. A given route was in one table or the other, based on the type of route. The default route was in the network routing table with a destination address of 0.0.0.0. There was a search made for a host route first, and if not found a search was made for a network route, and if still not found, a search was made for a default route. Only if all three failed was the destination unreachable. Section 11.5 of [Leffler et al. 1990] describes the hash table with linked lists used for the host and network routing tables.

Major changes took place in the internal representation of the routing table after the introduction of Reno [Sklower 1991]. These changes allow the same routing table function to support routing table for other protocol suites, notably the OSI protocols, which use variable-length addresses, unlike the fixed-length 32-bit Internet addresses. The internal representation of the routing table was changed, to provide faster lookups.

The Net/3 routing table uses a Patricia tree structure [Sedgewick 1990] to store both host addresses and network addresses. (Patricia stands for "Practical Algorithm for Rapidly Searching a Tree of Alphanumeric Information Coded in Alphanumeric.") The address being searched for and the nodes in the tree are considered as sequences of bits. This allows the same function to search one tree containing fixed-length 32-bit Internet addresses, another tree containing fixed-length 48-bit XNS addresses, and another tree containing variable-length OSI addresses.

The idea of using Patricia trees for the routing table is attributed to Vint Cerf and Robert E. Sklower [Sklower 1991]. These are actually binary radix tries with one-way branching.

An example is the easiest way to describe the algorithm. The goal of routing is to find the most specific address that matches the given destination: the search for the most specific implies that a host address is preferred over a network address.

over a default address.

Each entry has an associated network mask, although no mask is stored instead host routes have an implied mask of all one bits. An entry in the table matches a search key if the search key logically ANDed with the network mask equals the entry itself. A given search key might match multiple entries so with a single table for both network route and host routes, the table entries that more-specific routes are considered before less-specific routes.

Consider the examples in [Figure 18.3](#). The two search keys are 127.0.0.1 and 127.0.0.2 which we show in hexadecimal since the logical ANDing is easier to illustrate. The table entries are the host entry for 127.0.0.1 (with an implied mask of 0xffffffff) and the network entry for 127.0.0.0 (with a mask of 0xff000000).

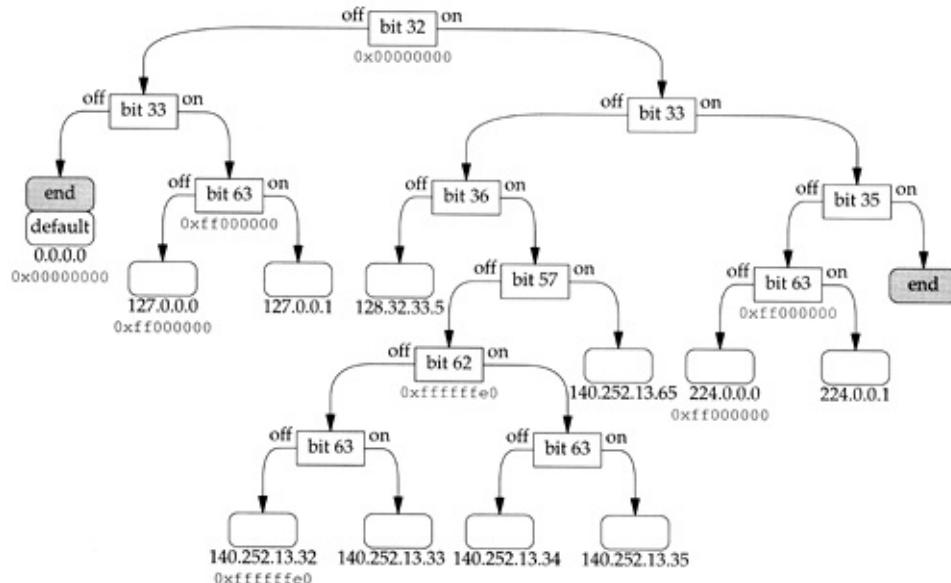
**Figure 18.3. Example routing table lookups for the two search keys 127.0.0.1 and 127.0.0.2.**

		search key = 127.0.0.1		search key = 127.0.0.2	
		host route	net route	host route	net route
1	search key	7f000001	7f000001	7f000002	7f000002
2	routing table key	7f000001	7f000000	7f000001	7f000000
3	routing table mask	ffffffffff	ff000000	ffffffffff	ff000000
4	logical AND of 1 and 3	7f000001	7f000000	7f000002	7f000000
	2 and 4 equal?	yes	yes	no	yes

Since the search key 127.0.0.1 matches both routing table entries, the table is organized so that the more-specific entry (127.0.0.1) is tried first.

[Figure 18.4](#) shows the internal representation of the Net/3 routing table corresponding to [Figure 18.2](#). This table was built from the output of the netstat command which dumps the tree structure of the routing tables.

**Figure 18.4. Net/3 routing table corresponding to Figure 18.2.**



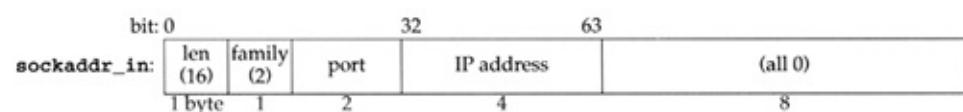
The two shaded boxes labeled "end" are leaves with special flags denoted tree. The left one has a key of all zero bits and the right one has a key two boxes stacked together at the left, labeled "end" and "default," are representation used for duplicate keys, which we describe in [Section 18](#)

The square-cornered boxes are called *internal nodes* or just *nodes*, and rounded corners are called *leaves*. Each internal node corresponds to a search key, and a branch is made to the left or the right. Each leaf corresponds to a host address or a network address. If there is a hexadecimal number b is a network address and the number specifies the network mask for the address. If there is a hexadecimal mask beneath a leaf node implies that the leaf is a host implied mask of 0xffffffff.

Some of the internal nodes also contain network masks, and we'll see how to use them for backtracking. Not shown in this figure is that every node also contains pointers to facilitate backtracking, deletion, and nonrecursive walks of the tree.

The bit comparisons are performed on socket address structures, so the bit offsets in Figure 18.4 are from the start of the socket address structure. Figure 18.5 shows the bit offsets for a `sockaddr_in` structure.

**Figure 18.5. Bit offsets in Internet socket address structure**



The highest-order bit of the IP address is at bit position 32 and the low position 63. We also show the length as 16 and the address family as 2 encounter these two values throughout our examples.

To work through the examples we also need to show the bit representations of addresses in the tree. These are shown in [Figure 18.6](#) along with some that are used in the examples that follow. The bit positions used in [Figure 18.6](#) points are shown in a bolder font.

**Figure 18.6. Bit representations of the IP addresses in Figure 18.5**

	32-bit IP address (bits 32–63)									dotted-decimal
bit:	3333	3333	4444	4444	4455	5555	5555	6666		
	2345	6789	0123	4567	8901	2345	6789	0123		
	0000	1010	0000	0001	0000	0010	0000	0011	10.1.2.3	
	0111	0000	0000	0000	0000	0000	0000	0001	112.0.0.1	
	0111	1111	0000	0000	0000	0000	0000	0000	127.0.0.0	
	0111	1111	0000	0000	0000	0000	0000	0001	127.0.0.1	
	0111	1111	0000	0000	0000	0000	0000	0011	127.0.0.3	
	1000	0000	0010	0000	0010	0001	0000	0101	128.32.33.5	
	1000	0000	0010	0000	0010	0001	0000	0110	128.32.33.6	
	1000	1100	1111	1100	0000	1101	0010	0000	140.252.13.32	
	1000	1100	1111	1100	0000	1101	0010	0001	140.252.13.33	
	1000	1100	1111	1100	0000	1101	0010	0010	140.252.13.34	
	1000	1100	1111	1100	0000	1101	0010	0011	140.252.13.35	
	1000	1100	1111	1100	0000	1101	0100	0001	140.252.13.65	
	1110	0000	0000	0000	0000	0000	0000	0000	224.0.0.0	
	1110	0000	0000	0000	0000	0000	0000	0001	224.0.0.1	

We now provide some specific examples of how the routing table search function works.

## ExampleHost Match

Assume the host address 127.0.0.1 is the search key. The destination address is 127.0.0.1. Bit 32 is off, so the left branch is made from the top of the tree. Bit 33 is on, so the right branch is made from the next node. Bit 63 is on, so the right branch is made from the next node. This next node is a leaf, so the search key (127.0.0.1) is compared with the leaf (127.0.0.1). They match exactly so this routing table entry is returned by the search function.

## ExampleHost Match

Next assume the search key is the address 140.252.13.35. Bit 32 is on

made from the top of the tree. Bit 33 is off, bit 36 is on, bit 57 is off, bit 60 is on, so the search ends at the leaf on the bottom labeled 140.252.13 which matches the routing table key exactly.

### ExampleNetwork Match

The search key is 127.0.0.2. Bit 32 is off, bit 33 is on, and bit 63 is off at the leaf labeled 127.0.0.0. The search key and the routing table key are equal, so a network match is tried. The search key is logically ANDed with the mask (0xff000000) and since the result equals the routing table key, this entry is a match.

### ExampleDefault Match

The search key is 10.1.2.3. Bit 32 is off and bit 33 is off, so the search ends with the duplicate keys labeled "end" and "default." The routing table key in these two leaves is 0.0.0.0. The search key and the routing table key are equal, so a network match is tried. This match is tried for all duplicate keys that have a network mask. The first key (the end marker) doesn't have a network mask, so the second key (the default entry) has a mask of 0x00000000. The search key is logically ANDed with this mask and since the result equals the routing table key (0), this entry is a match. The default route is used.

### ExampleNetwork Match with Backtracking

The search key is 127.0.0.3. Bit 32 is off, bit 33 is on, and bit 63 is on, so the search ends at the leaf labeled 127.0.0.1. The search key and the routing table key are equal, so a network match cannot be attempted since this leaf does not have a network mask. Backtracking now takes place.

The backtracking algorithm is to move up the tree, one level at a time. When the search key is compared against a node that contains a mask, the search key is logically ANDed with the mask. Another search is made of the subtree starting at the node with the mask. If a match is found, the search continues with the ANDed key. If a match isn't found, the backtracking continues until the top is reached.

In this example the search moves up one level to the node for bit 63 and bit 60. The search key is logically ANDed with the mask (0xff000000) to get a key of 127.0.0.0. Another search is made starting at this node for 127.0.0.0. The left branch is taken to the leaf labeled 127.0.0.0. The new search key and the routing table key are equal, so this leaf is the match.

## ExampleBacktracking Multiple Levels

The search key is 112.0.0.1. Bit 32 is off, bit 33 is on, and bit 63 is on, at the leaf labeled 127.0.0.1. The keys are not equal and the routing table has a network mask, so backtracking takes place.

The search moves up one level to the node for bit 63, which contains a mask logically ANDed with the mask of 0xff000000 and another search is made to the right branch. Bit 63 is off in the new search key, so the left branch is made to 127.0.0.0. A comparison is made but the ANDed search key (112.0.0.0) does not match the search key in the table.

Backtracking continues up one level from the bit-63 node to the bit-33 node. Since the bit-33 node does not have a mask, so the backtracking continues upward. The next node is the bit-32 node and it has a mask. The search key (112.0.0.1) is logically ANDed with the mask (0x00000000) and a new search started from that point. Bit 32 is off if the search key is 0x00000000 and bit 33 is on, so the search ends up at the leaf labeled "end" and "default". Since no duplicate keys are traversed and the default key matches the new search key, the default route is used.

As we can see in this example, if a default route is present in the routing table, the backtracking ends up at the top node in the tree, its mask is all zero bits, so the search continues to proceed to the leftmost leaf in the tree for a match with the search key.

## ExampleHost Match with Backtracking and Cloning

The search key is 224.0.0.5. Bit 32 is on, bit 33 is on, bit 35 is off, and the search ends up at the leaf labeled 224.0.0.1. This routing table key does not match the search key, and the routing table entry does not contain a network mask, so backtracking takes place.

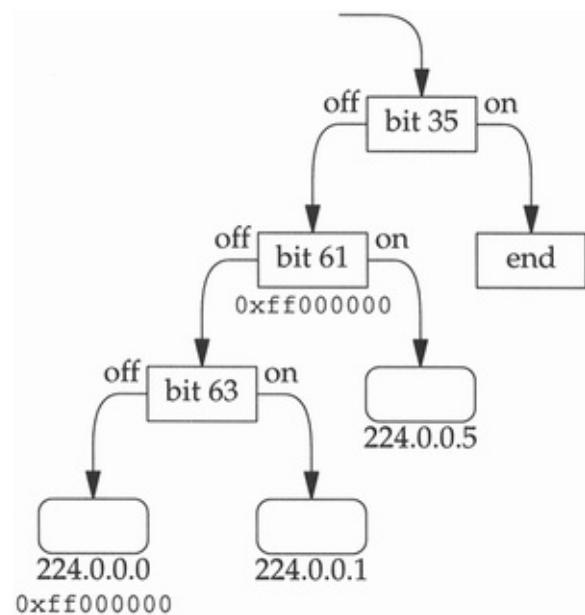
The backtracking moves one level up to the node that tests bit 63. This node has a mask of 0xff000000, so the search key ANDed with the mask yields a new search key (0x00000000). Another search is made, starting at this node. Since bit 63 is off in the search key, the right branch is taken to the leaf labeled 224.0.0.0. This routing table key matches the search key, so this entry is a match.

This route has the "clone" flag set ([Figure 18.2](#)), so a new leaf is created for the search key 224.0.0.5. The new routing table entry is

Destination	Gateway	Flags	Refs
224.0.0.5	link#1	UHL	0

and [Figure 18.7](#) shows the new arrangement of the right side of the root of [Figure 18.4](#), starting with the node for bit 35. Notice that whenever a route is inserted into the tree, two nodes are needed: one for the leaf and one for the internal node corresponding to the bit to test.

**Figure 18.7. Modification of Figure 18.4 after inserting entry**

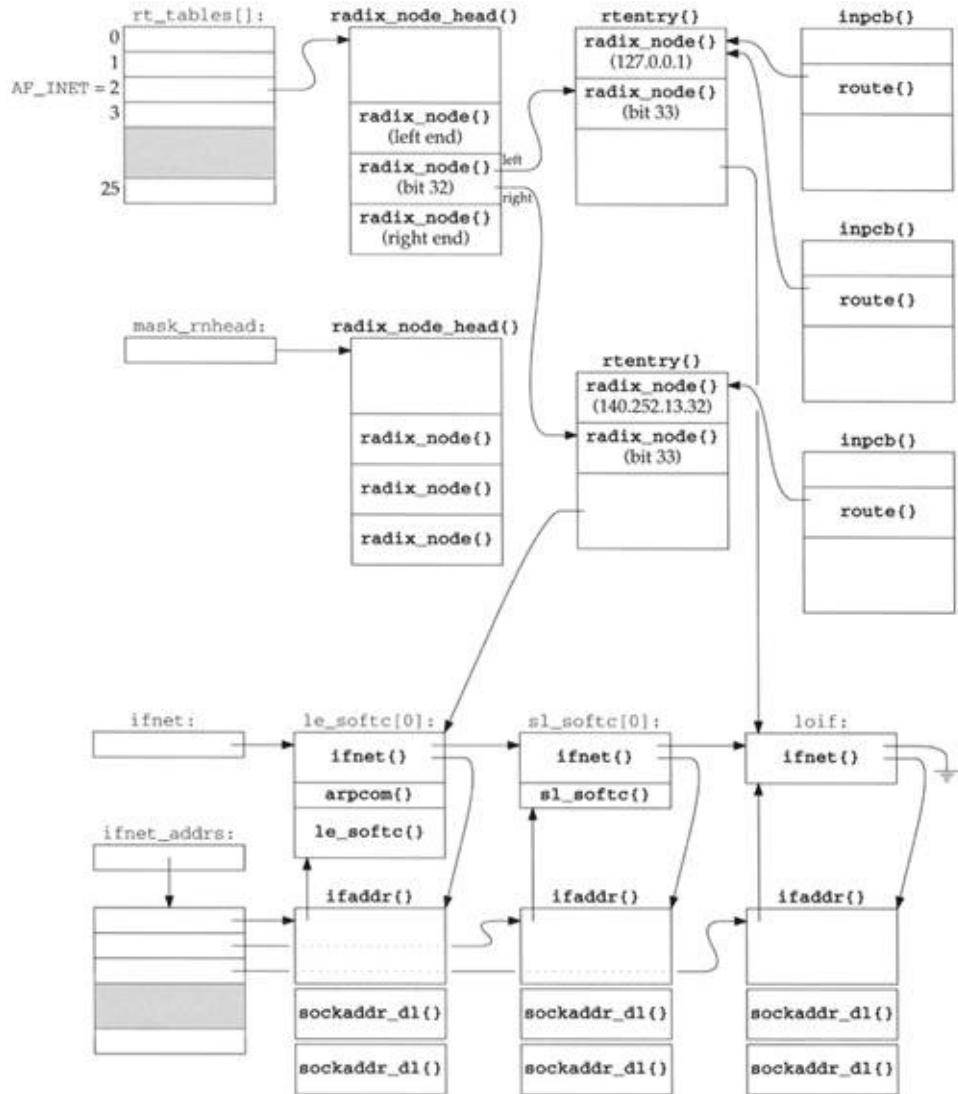


This newly created entry is the one returned to the caller who was searching for it.

## The Big Picture

[Figure 18.8](#) shows a bigger picture of all the data structures involved. The information in this figure is from [Figure 3.32](#).

**Figure 18.8. Data structures involved with routing tables**



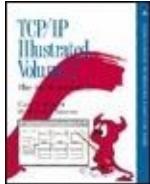
There are numerous points about this figure that we'll note now and de this chapter.

- `rt_tables` is an array of pointers to `radix_node_head` structures. Th array for each address family. `rt_tables[AF_INET]` points to the top routing table tree.
- The `radix_node_head` structure contains three `radix_node` structur are built when the tree is initialized and the middle of the three is t This corresponds to the top box in [Figure 18.4](#), labeled "bit 32." Th `radix_node` structures is the leftmost leaf in [Figure 18.4](#) (the share default route) and the third of the three is the rightmost leaf. An e consists of just these three `radix_node` structures; we'll see how it rn\_inithead function.

- The global mask\_rnhead also points to a radix\_node\_head structure, which points to a separate tree of all the masks. Notice in [Figure 18.4](#) that of the eight entries in the tree, one is duplicated four times and two are duplicated once. By keeping multiple pointers to the same mask, only one copy of each unique mask is maintained.
- The routing table tree is built from rtentry structures, and we show the structure in [Figure 18.8](#). Each rtentry structure contains two radix\_node structures. Every time a new entry is inserted into the tree, two nodes are required: one corresponding to a bit to be tested, and a leaf node corresponding to the network route. In each rtentry structure we also show which bit to test, which bit corresponds to and the address contained in the leaf node.

The remainder of the rtentry structure is the focal point of information. We show only a single pointer from this structure to the corresponding route, but this structure also contains a pointer to the ifaddr structure, a pointer to the next rtentry structure if this entry is an indirect entry for the route, a pointer to the previous entry for the route, and so on.

- Protocol control blocks ([Chapter 22](#)), of which one exists for each local interface ([Figure 22.1](#)), contain a route structure that points to an rtentry structure. The TCP output functions both pass a pointer to the route structure in the argument to ip\_output, each time an IP datagram is sent. PCBs that have multiple interfaces point to the same routing table entry.
-



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.3 Routing Sockets

When the routing table changes were made with 4.3BSD Reno, the interaction of processes with the routing subsystem also changed—the concept of routing sockets was introduced. Prior to 4.3BSD Reno, fixed-length ioctl s were issued by a process (such as the route command) to modify the routing table. 4.3BSD Reno changed this to a more generalized message-passing scheme using the new PF\_ROUTE domain. A process creates a raw socket in the PF\_ROUTE domain and can send routing messages to the kernel, and receives routing messages from the kernel (e.g., redirects and other asynchronous

notifications from the kernel).

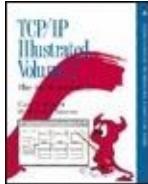
[Figure 18.9](#) shows the 12 different types of routing messages. The message type is the `rtm_type` field in the `rt_msghdr` structure, which we describe in [Figure 19.16](#). Only five of the messages can be issued by a process (a write to a routing socket), but all 12 can be received by a process.

## Figure 18.9. Types of messages exchanged across a routing socket.

<code>rtm_type</code>	To kernel?	From kernel?	Description	Structure type
<code>RTM_ADD</code>	•	•	add route	<code>rt_msghdr</code>
<code>RTM_CHANGE</code>	•	•	change gateway, metrics, or flags	<code>rt_msghdr</code>
<code>RTM_DELADDR</code>		•	address being removed from interface	<code>ifa_msghdr</code>
<code>RTM_DELETE</code>	•	•	delete route	<code>rt_msghdr</code>
<code>RTM_GET</code>	•	•	report metrics and other route information	<code>rt_msghdr</code>
<code>RTM_IFINFO</code>		•	interface going up, down, etc.	<code>if_msghdr</code>
<code>RTM_LOCK</code>	•	•	lock specified metrics	<code>rt_msghdr</code>
<code>RTM_LOSING</code>		•	kernel suspects route is failing	<code>rt_msghdr</code>
<code>RTM_MISS</code>		•	lookup failed on this address	<code>rt_msghdr</code>
<code>RTM_NEWADDR</code>		•	address being added to interface	<code>ifa_msghdr</code>
<code>RTM_REDIRECT</code>		•	kernel told to use different route	<code>rt_msghdr</code>
<code>RTM_RESOLVE</code>		•	request to resolve destination to link-layer address	<code>rt_msghdr</code>

We'll defer our discussion of these routing messages until [Chapter 19](#).

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.4 Code Introduction

Three headers and five C files define the various structures and functions used for routing. These are summarized in [Figure 18.10](#).

**Figure 18.10. Files discussed in this chapter.**

File	Description
net/radix.h	radix node definitions
net/raw_cb.h	routing control block definitions
net/route.h	routing structures
net/radix.c	radix node (Patricia tree) functions
net/raw_cb.c	routing control block functions
net/raw_usrreq.c	routing control block functions
net/route.c	routing functions
net/rtsock.c	routing socket functions

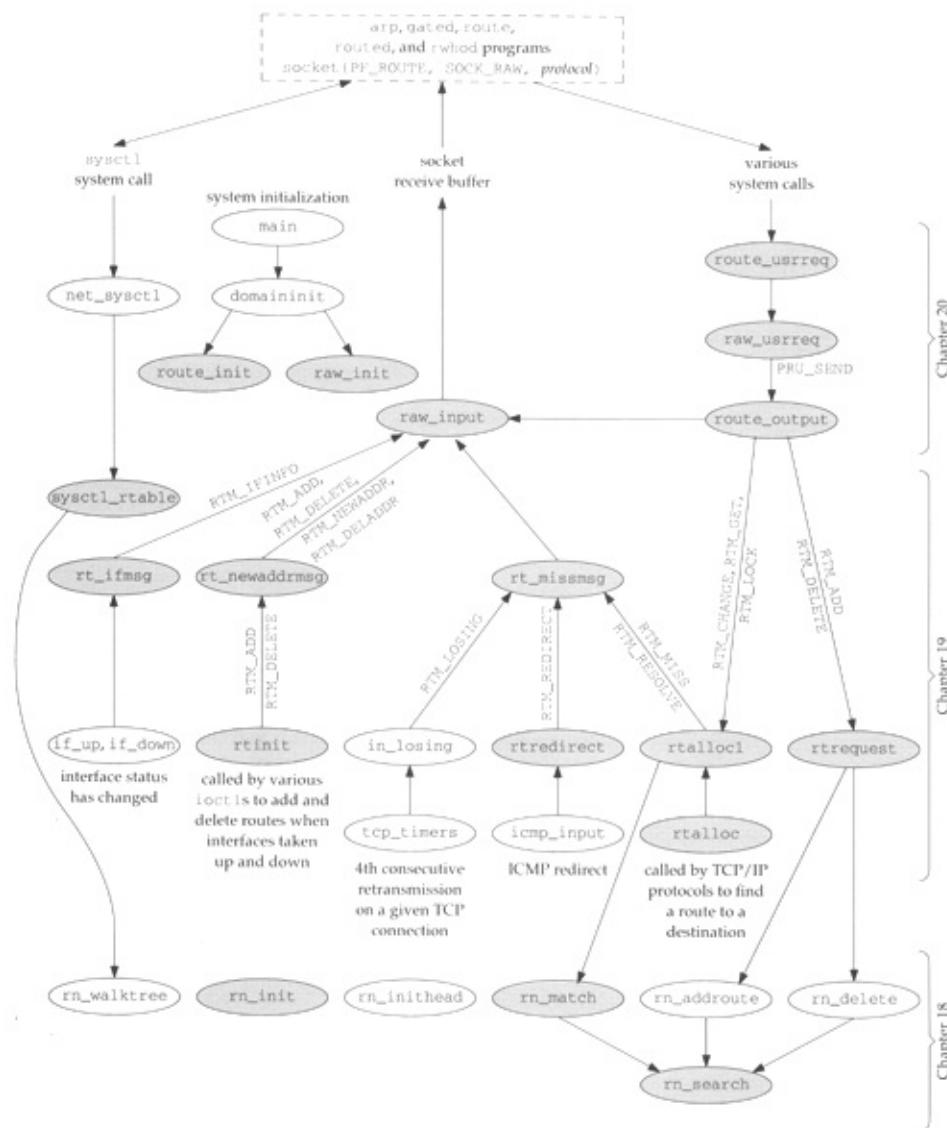
In general, the prefix rn\_ denotes the radix node functions that search and manipulate the Patricia trees, the raw\_ prefix denotes the routing control block functions, and the three prefixes route\_, rt\_, and rt denote the general routing functions.

We use the term *routing control blocks* instead of *raw control blocks* in all the routing chapters, even though the files and functions begin with the prefix raw. This is to avoid confusion with the raw IP control blocks and functions, which we discuss in [Chapter 32](#). Although the raw control blocks and their associated functions are used for more than just routing sockets in Net/3 (one of the raw OSI protocols uses these structures and functions), our use in this text is only with routing sockets in the PF\_ROUTE domain.

[Figure 18.11](#) shows the primary routing functions and their relationships. The shaded ellipses are the ones we cover in this chapter and the next two. We also show where each of the 12 routing

message types are generated.

**Figure 18.11. Relationships between the various routing functions.**



**rtalloc** is the function called by the Internet protocols to look up routes to

destinations. We've already encountered `rtalloc` in the `ip_rtaddr`, `ip_forward`, `ip_output`, and `ip_setmoptions` functions. We'll also encounter it later in the `in_pcboconnect` and `tcp_mss` functions.

We also show in [Figure 18.11](#) that five programs typically create sockets in the routing domain:

- `arp` manipulates the ARP cache, which is stored in the IP routing table in Net/3 ([Chapter 21](#)),
- `gated` and `routed` are routing daemons that communicate with other routers and manipulate the kernel's routing table as the routing environment changes (routers and links go up or down),
- `route` is a program typically executed by start-up scripts or by the system administrator to add or delete routes, and
- `rwhod` issues a routing sysctl on start-up to determine the attached

interfaces.

Naturally, any process (with superuser privilege) can open a routing socket to send and receive messages to and from the routing subsystem; we show only the common system programs in [Figure 18.11](#).

## Global Variables

The global variables introduced in the three routing chapters are shown in [Figure 18.12](#).

**Figure 18.12. Global variables in the three routing chapters.**

Variable	Datatype	Description
rt_tables	struct radix_node_head * [1]	array of pointers to heads of routing tables
mask_rnhead	struct radix_node_head *	pointer to head of mask table
rn_mkfreelist	struct radix_mask *	head of linked list of available radix_mask structures
max_keylen	int	longest routing table key, in bytes
rn_zeros	char *	array of all zero bits, of length max_keylen
rn_ones	char *	array of all one bits, of length max_keylen
maskedKey	char *	array for masked search key, of length max_keylen
rtstat	struct rtstat	routing statistics (Figure 18.13)
rttrash	int	#routes not in table but not freed
rawcb	struct rawcb	head of doubly linked list of routing control blocks
raw_recvspace	u_long	default size of routing socket receive buffer, 8192 bytes
raw_sendspace	u_long	default size of routing socket send buffer, 8192 bytes
route_cb	struct route_cb	#routing socket listeners, per protocol, and total
route_dst	struct sockaddr	temporary for destination of routing message
route_src	struct sockaddr	temporary for source of routing message
route_proto	struct sockproto	temporary for protocol of routing message

## Statistics

Some routing statistics are maintained in the global structure `rtstat`, described in Figure 18.13.

**Figure 18.13. Routing statistics maintained in the `rtstat` structure.**

rtstat member	Description	Used by SNMP
<code>rts_badredirect</code>	#invalid redirect calls	
<code>rts_dynamic</code>	#routes created by redirects	
<code>rts_newgateway</code>	#routes modified by redirects	
<code>rts_unreach</code>	#lookups that failed	
<code>rts_wildcard</code>	#lookups matched by wildcard (never used)	

We'll see where these counters are incremented as we proceed through the code. None are used by SNMP.

Figure 18.14 shows some sample output of these statistics from the `netstat -rs` command, which displays this structure.

**Figure 18.14. Sample routing statistics.**

netstat -rs output	rtstat member
1029 bad routing redirects 0 dynamically created routes 0 new gateways due to redirects 0 destinations found unreachable 0 uses of a wildcard route	rts_badredirect rts_dynamic rts_newgateway rts_unreach rts_wildcard

## SNMP Variables

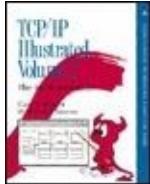
Figure 18.15 shows the IP routing table, named ipRouteTable, and the kernel variables that supply the corresponding value.

**Figure 18.15. IP routing table:  
ipRouteTable.**

IP routing table, index = <ipRouteDest>		
SNMP variable	Variable	Description
ipRouteDest	rt_key	Destination IP address. A value of 0.0.0.0 indicates a default entry.
ipRouteIfIndex	rt_ifp.if_index	Interface number: ifIndex.
ipRouteMetric1	-1	Primary routing metric. The meaning of the metric depends on the routing protocol (ipRouteProto). A value of -1 means it is not used.
ipRouteMetric2	-1	Alternative routing metric.
ipRouteMetric3	-1	Alternative routing metric.
ipRouteMetric4	-1	Alternative routing metric.
ipRouteNextHop	rt_gateway (see text)	IP address of next-hop router.
ipRouteType	(see text)	Route type: 1 = other, 2 = invalidated route, 3 = direct, 4 = indirect.
ipRouteProto	(see text)	Routing protocol: 1 = other, 4 = ICMP redirect, 8 = RIP, 13 = OSPF, 14 = BGP, and others.
ipRouteAge	(not implemented)	Number of seconds since route was last updated or determined to be correct.
ipRouteMask	rt_mask	Mask to be logically ANDed with destination IP address before being compared with ipRouteDest.
ipRouteMetric5	-1	Alternative routing metric.
ipRouteInfo	NULL	Reference to MIB definitions specific to this particular routing protocol.

For `ipRouteType`, if the `RTF_GATEWAY` flag is set in `rt_flags`, the route is remote (4); otherwise the route is direct (3). For `ipRouteProto`, if either the `RTF_DYNAMIC` or `RTF_MODIFIED` flag is set, the route was created or modified by ICMP (4), otherwise the value is other (1). Finally, if the `rt_mask` pointer is null, the returned mask is all one bits (i.e., a host route).

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.5 Radix Node Data Structures

In [Figure 18.8](#) we see that the head of each routing table is a `radix_node_head` and all the nodes in the routing tree, both the internal nodes and the leaves, are `radix_node` structures. The `radix_node_head` structure is shown in [Figure 18.16](#).

**Figure 18.16. `radix_node_head` structure:  
the top of each routing tree.**

```

91 struct radix_node_head {
92     struct radix_node *rnh_treetop;
93     int     rnh_addrsize;      /* (not currently used) */
94     int     rnh_pktsize;      /* (not currently used) */
95     struct radix_node *(*rnh_addaddr) /* add based on sockaddr */
96         (void *v, void *mask,
97          struct radix_node_head * head, struct radix_node nodes[]);
98     struct radix_node *(*rnh_addpkt) /* add based on packet hdr */
99         (void *v, void *mask,
100          struct radix_node_head * head, struct radix_node nodes[]);
101    struct radix_node *(*rnh_deladdr) /* remove based on sockaddr */
102        (void *v, void *mask, struct radix_node_head * head);
103    struct radix_node *(*rnh_delpkt) /* remove based on packet hdr */
104        (void *v, void *mask, struct radix_node_head * head);
105    struct radix_node *(*rnh_matchaddr) /* locate based on sockaddr */
106        (void *v, struct radix_node_head * head);
107    struct radix_node *(*rnh_matchpkt) /* locate based on packet hdr */
108        (void *v, struct radix_node_head * head);
109    int     (*rnh_walktree) /* traverse tree */
110        (struct radix_node_head * head, int (*f) (), void *w);
111    struct radix_node rnh_nodes[3]; /* top and end nodes */
112 };

```

92

rnh\_treetop points to the top radix\_node structure for the routing tree. Notice that three of these structures are allocated at the end of the radix\_node\_head, and the middle one of these is initialized as the top of the tree ([Figure 18.8](#)).

93-94

rnh\_addrsize and rnh\_pktsize are not currently used.

rnh\_addrsize is to facilitate porting the routing table code to systems that don't have a length byte in the socket address structure. rnh\_pktsize is to

allow using the radix node machinery to examine addresses in packet headers without having to copy the address into a socket address structure.

## 95-110

The seven function pointers, `rnh_addaddr` through `rnh_walktree`, point to functions that are called to operate on the tree. Only four of these pointers are initialized by `rn_inithead` and the other three are never used by Net/3, as shown in [Figure 18.17](#).

**Figure 18.17. The seven function pointers in the `radix_node_head` structure.**

Member	Initialized to (by <code>rn_inithead</code> )
<code>rnh_addaddr</code>	<code>rn_addroute</code>
<code>rnh_addpkt</code>	<code>NULL</code>
<code>rnh_deladdr</code>	<code>rn_delete</code>
<code>rnh_delpkt</code>	<code>NULL</code>
<code>rnh_matchaddr</code>	<code>rn_match</code>
<code>rnh_matchpkt</code>	<code>NULL</code>
<code>rnh_walktree</code>	<code>rn_walktree</code>

## 111-112

[Figure 18.18](#) shows the radix\_node structure that forms the nodes of the tree. In [Figure 18.8](#) we see that three of these are allocated in the radix\_node\_head and two are allocated in each rentry structure.

## Figure 18.18. radix\_node structure: the nodes of the routing tree.

```
----- radix.h
40 struct radix_node {
41     struct radix_mask *rn_mklist; /* list of masks contained in subtree */
42     struct radix_node *rn_p;    /* parent pointer */
43     short   rn_b;             /* bit offset: -1-index(netmask) */
44     char    rn_bmask;         /* node: mask for bit test */
45     u_char  rn_flags;        /* Figure 18.20 */
46     union {
47         struct {           /* leaf only data: rn_b < 0 */
48             caddr_t rn_Key; /* object of search */
49             caddr_t rn_Mask; /* netmask, if present */
50             struct radix_node *rn_Dupedkey;
51         } rn_leaf;
52         struct {           /* node only data: rn_b >= 0 */
53             int    rn_Off;  /* where to start compare */
54             struct radix_node *rn_L; /* left pointer */
55             struct radix_node *rn_R; /* right pointer */
56         } rn_node;
57     } rn_u;
58 };
59 #define rn_dupedkey rn_u.rn_leaf.rn_Dupedkey
60 #define rn_key      rn_u.rn_leaf.rn_Key
61 #define rn_mask     rn_u.rn_leaf.rn_Mask
62 #define rn_off      rn_u.rn_node.rn_Off
63 #define rn_l       rn_u.rn_node.rn_L
64 #define rn_r       rn_u.rn_node.rn_R
----- radix.h
```

41-45

The first five members are common to both internal nodes and leaves, followed by a union defining three members if the node is a leaf, or a different three

members if the node is internal. As is common throughout the Net/3 code, a set of #define statements provide shorthand names for the members in the union.

41-42

`rn_mklist` is the head of a linked list of masks for this node. We describe this field in [Section 18.9](#). `rn_p` points to the parent node.

43

If `rn_b` is greater than or equal to 0, the node is an internal node, else the node is a leaf. For the internal nodes, `rn_b` is the bit number to test: for example, its value is 32 in the top node of the tree in [Figure 18.4](#). For leaves, `rn_b` is negative and its value is -1 minus the *index of the network mask*. This index is the first bit number where a 0 occurs. [Figure 18.19](#) shows the indexes of the masks from [Figure 18.4](#).

**Figure 18.19. Example of mask indexes.**

	32-bit IP mask (bits 32–63)									index	rn_b
	3333 3333 4444 4444 4455 5555 5555 6666 2345 6789 0123 4567 8901 2345 6789 0123										
00000000:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0	-1
ff000000:	1111	1111	0000	0000	0000	0000	0000	0000	0000	40	-41
fffffe0:	1111	1111	1111	1111	1111	1111	1110	0000	0000	59	-60

As we can see, the index of the all-zero mask is handled specially: its index is 0, not 32.

44

rn\_bmask is a 1-byte mask used with the internal nodes to test whether the corresponding bit is on or off. Its value is 0 in leaves. We'll see how this member is used with the rn\_off member shortly.

45

[Figure 18.20](#) shows the three values for the rn\_flags member.

## Figure 18.20. rn\_flags values.

Constant	Description
RNF_ACTIVE	this node is alive (for rtfree)
RNF_NORMAL	leaf contains normal route (not currently used)
RNF_ROOT	node is in the radix_node_head structure

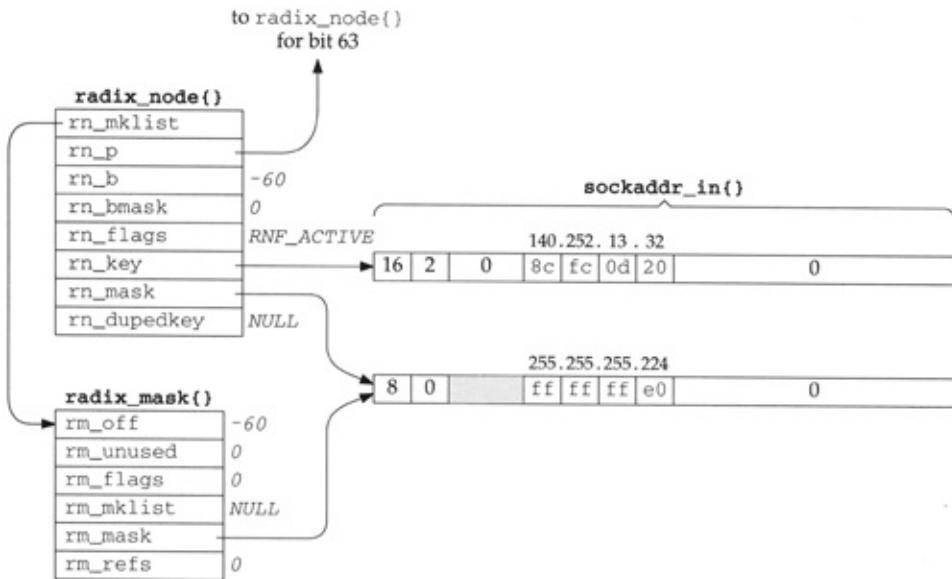
The RNF\_ROOT flag is set only for the three radix nodes in the radix\_node\_head structure: the top of the tree and the left and right end nodes. These three nodes can never be deleted from the routing tree.

48-49

For a leaf, rn\_key points to the socket address structure and rn\_mask points to a socket address structure containing the mask. If rn\_mask is null, the implied mask is all one bits (i.e., this route is to a host, not to a network).

[Figure 18.21](#) shows an example corresponding to the leaf for 140.252.13.32 in [Figure 18.4](#).

**Figure 18.21. radix\_node structure corresponding to leaf for 140.252.13.32 in Figure 18.4.**



This example also shows a `radix_mask` structure, which we describe in [Figure 18.22](#). We draw this latter structure with a smaller width, to help distinguish it as a different structure from the `radix_node`; we'll encounter both structures in many of the figures that follow. We describe the reason for the `radix_mask` structure in [Section 18.9](#).

**Figure 18.22. radix\_mask structure.**

---

```

76 extern struct radix_mask {
77     short    rm_b;           /* bit offset; -1-index(netmask) */
78     char     rm_unused;      /* cf. rn_bmask */
79     u_char   rm_flags;       /* cf. rn_flags */
80     struct radix_mask *rm_mklist; /* more masks to try */
81     caddr_t  rm_mask;        /* the mask */
82     int      rm_refs;        /* # of references to this struct */
83 } *rn_mkfreelist;

```

---

The rn\_b of 60 corresponds to an index of 59. rn\_key points to a sockaddr\_in, with a length of 16 and an address family of 2 (AF\_INET). The mask structure pointed to by rn\_mask and rm\_mask has a length of 8 and a family of 0 (this family is AF\_UNSPEC, but it is never even looked at).

## 50-51

The rn\_dupedkey pointer is used when there are multiple leaves with the same key. We describe these in [Section 18.9](#).

## 52-58

We describe rn\_off in [Section 18.8](#). rn\_1 and rn\_r are the left and right pointers for the internal node.

[Figure 18.22](#) shows the radix\_mask structure.

## 76-83

Each of these structures contains a pointer to a mask: rm\_mask, which is really a pointer to a socket address structure

containing the mask. Each radix\_node structure points to a linked list of radix\_mask structures, allowing multiple masks per node: rn\_mklist points to the first, and then each rm\_mklist points to the next. This structure definition also declares the global rn\_mkfreelist, which is the head of a linked list of available structures.

---

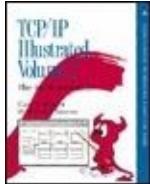
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.6 Routing Structures

The focal points of access to the kernel's routing information are

**1. the `rtalloc` function, which searches for a route to a destination,**

- the route structure that is filled in by this function, and
- the rtentry structure that is pointed to by the route structure.

[Figure 18.8](#) showed that the protocol control blocks (PCBs) used by UDP and TCP ([Chapter 22](#)) contain a route structure, which we show in [Figure 18.23](#).

## Figure 18.23. route structure.

```
46 struct route {  
47     struct rtentry *ro_rt;      /* pointer to struct with information */  
48     struct sockaddr ro_dst;    /* destination of this route */  
49 };
```

route.h

route.h

ro\_dst is declared as a generic socket address structure, but for the Internet protocols it is a sockaddr\_in. Notice that unlike most references to this type of structure, ro\_dst is the structure itself, not a pointer to one.

At this point it is worth reviewing [Figure 8.24](#), which shows the use of these routes every time an IP datagram is output.

- If the caller passes a pointer to a route structure, that structure is used. Otherwise a local route structure is used and it is set to 0, setting ro\_rt to a null pointer. UDP and TCP pass a pointer to the route structure in their PCB to ip\_output.
- If the route structure points to an rtentry structure (the ro\_rt pointer is nonnull), and if the referenced interface

is still up, and if the destination address in the route structure equals the destination address of the IP datagram, that route is used. Otherwise the socket address structure `ro_dst` is filled in with the destination IP address and `rtalloc` is called to locate a route to that destination. For a TCP connection the destination address of the datagram never changes from the destination address of the route, but a UDP application can send a datagram to a different destination with each `sendto`.

- If `rtalloc` returns a null pointer in `ro_rt`, a route was not found and `ip_output` returns an error.
- If the `RTF_GATEWAY` flag is set in the `rtentry` structure, the route is indirect (the `G` flag in [Figure 18.2](#)). The destination address (`dst`) for the interface output function becomes the IP address of the gateway, the `rt_gateway` member, not the destination address of the IP datagram.

[Figure 18.24](#) shows the `rtentry` structure.

## Figure 18.24. rtentry structure.

```
83 struct rtentry {
84     struct radix_node rt_nodes[2]; /* a leaf and an internal node */
85     struct sockaddr *rt_gateway; /* value associated with rn_key */
86     short rt_flags; /* Figure 18.25 */
87     short rt_refcnt; /* #held references */
88     u_long rt_use; /* raw #packets sent */
89     struct ifnet *rt_ifp; /* interface to use */
90     struct ifaddr *rt_ifa; /* interface address to use */
91     struct sockaddr *rt_genmask; /* for generation of cloned routes */
92     caddr_t rt_llinfo; /* pointer to link level info cache */
93     struct rt_metrics rt_rmx; /* metrics: Figure 18.26 */
94     struct rtentry *rt_gwroute; /* implied entry for gatewayed routes */
95 };
96 #define rt_key(r) ((struct sockaddr *)((r)->rt_nodes->rn_key))
97 #define rt_mask(r) ((struct sockaddr *)((r)->rt_nodes->rn_mask))
```

83-84

Two radix\_node structures are contained within this structure. As we noted in the example with [Figure 18.7](#), each time a new leaf is added to the routing tree a new internal node is also added. rt\_nodes[0] contains the leaf entry and rt\_nodes[1] contains the internal node. The two #define statements at the end of [Figure 18.24](#) provide a shorthand access to the key and mask of this leaf node.

86

[Figure 18.25](#) shows the various constants stored in rt\_flags and the corresponding character output by netstat in the "Flags"

column (Figure 18.2).

**Figure 18.25. rt\_flags values.**

Constant	netstat flag	Description
<i>RTF_BLACKHOLE</i>		discard packets without error (loopback driver: Figure 5.27)
<i>RTF_CLONING</i>	C	generate new routes on use (used by ARP)
<i>RTF_DONE</i>	d	kernel confirmation that message from process was completed
<i>RTF_DYNAMIC</i>	D	created dynamically (by redirect)
<i>RTF_GATEWAY</i>	G	destination is a gateway (indirect route)
<i>RTF_HOST</i>	H	host entry (else network entry)
<i>RTF_LLINFO</i>	L	set by ARP when <i>rt_llinfo</i> pointer valid
<i>RTF_MASK</i>	m	subnet mask present (not used)
<i>RTF_MODIFIED</i>	M	modified dynamically (by redirect)
<i>RTF_PROTO1</i>	1	protocol-specific routing flag
<i>RTF_PROTO2</i>	2	protocol-specific routing flag (ARP uses)
<i>RTF_REJECT</i>	R	discard packets with error (loopback driver: Figure 5.27)
<i>RTF_STATIC</i>	S	manually added entry (route program)
<i>RTF_UP</i>	U	route usable
<i>RTF_XRESOLVE</i>	X	external daemon resolves name (used with X.25)

The RTF\_BLACKHOLE flag is not output by netstat and the two with lowercase flag characters, RTF\_DONE and RTF\_MASK, are used in routing messages and not normally stored in the routing table entry.

85

If the RTF\_GATEWAY flag is set, *rt\_gateway* contains a pointer to a socket address structure containing the address (e.g., the IP address) of that gateway. Also, *rt\_gwroute* points to the *rtentry* for that gateway. This latter pointer was used

in ether\_output ([Figure 4.15](#)).

87

rt\_refcnt counts the "held" references to this structure. We describe this counter at the end of [Section 19.3](#). This counter is output as the "Refs" column in [Figure 18.2](#).

88

rt\_use is initialized to 0 when the structure is allocated; we saw it incremented in [Figure 8.24](#) each time an IP datagram was output using the route. This counter is also the value printed in the "Use" column in [Figure 18.2](#).

89-90

rt\_ifp and rt\_ifa point to the interface structure and the interface address structure, respectively. Recall from [Figure 6.5](#) that a given interface can have multiple addresses, so minimally the rt\_ifa is required.

92

The `rt_llinfo` pointer allows link-layer protocols to store pointers to their protocol-specific structures in the routing table entry. This pointer is normally used with the `RTF_LLINFO` flag. [Figure 21.1](#) shows how ARP uses this pointer.

93

[Figure 18.26](#) shows the `rt_metrics` structure, which is contained within the `rtentry` structure. [Figure 27.3](#) shows that TCP uses six members in this structure.

## [Figure 18.26. `rt\_metrics` structure.](#)

```
54 struct rt_metrics {
55     u_long rmx_locks;           /* bitmask for values kernel leaves alone */
56     u_long rmx_mtu;            /* MTU for this path */
57     u_long rmx_hopcount;       /* max hops expected */
58     u_long rmx_expire;         /* lifetime for route, e.g. redirect */
59     u_long rmx_recvpipe;        /* inbound delay-bandwidth product */
60     u_long rmx_sendpipe;        /* outbound delay-bandwidth product */
61     u_long rmx_ssthresh;       /* outbound gateway buffer limit */
62     u_long rmx_rtt;             /* estimated round trip time */
63     u_long rmx_rttvar;          /* estimated RTT variance */
64     u_long rmx_pktsent;         /* #packets sent using this route */
65 };
```

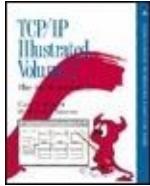
54-65

`rmx_locks` is a bitmask telling the kernel which of the eight metrics that follow must not be modified. The values for this

bitmask are shown in [Figure 20.13](#).

`rmx_expire` is used by ARP ([Chapter 21](#)) as a timer for each ARP entry. Contrary to the comment with `rmx_expire`, it is not used for redirects.

[Figure 18.28](#) summarizes the structures that we've described, their relationships, and the various types of socket address structures they reference. The `rtentry` that we show is for the route to 128.32.33.5 in [Figure 18.2](#). The other `radix_node` contained in the `rtentry` is for the bit 36 test right above this node in [Figure 18.4](#). The two `sockaddr_dl` structures pointed to by the first `ifaddr` were shown in [Figure 3.38](#). Also note from [Figure 6.5](#) that the `ifnet` structure is contained within an `le_softc` structure, and the second `ifaddr` structure is contained within an `in_ifaddr` structure.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.7 Initialization: `route_init` and `rtable_init` Functions

The initialization of the routing tables is somewhat obscure and takes us back to the domain structures in [Chapter 7](#). Before outlining the function calls, [Figure 18.27](#) shows the relevant fields from the domain structure ([Figure 7.5](#)) for various protocol families.

**Figure 18.27. Members of domain structure relevant to routing.**

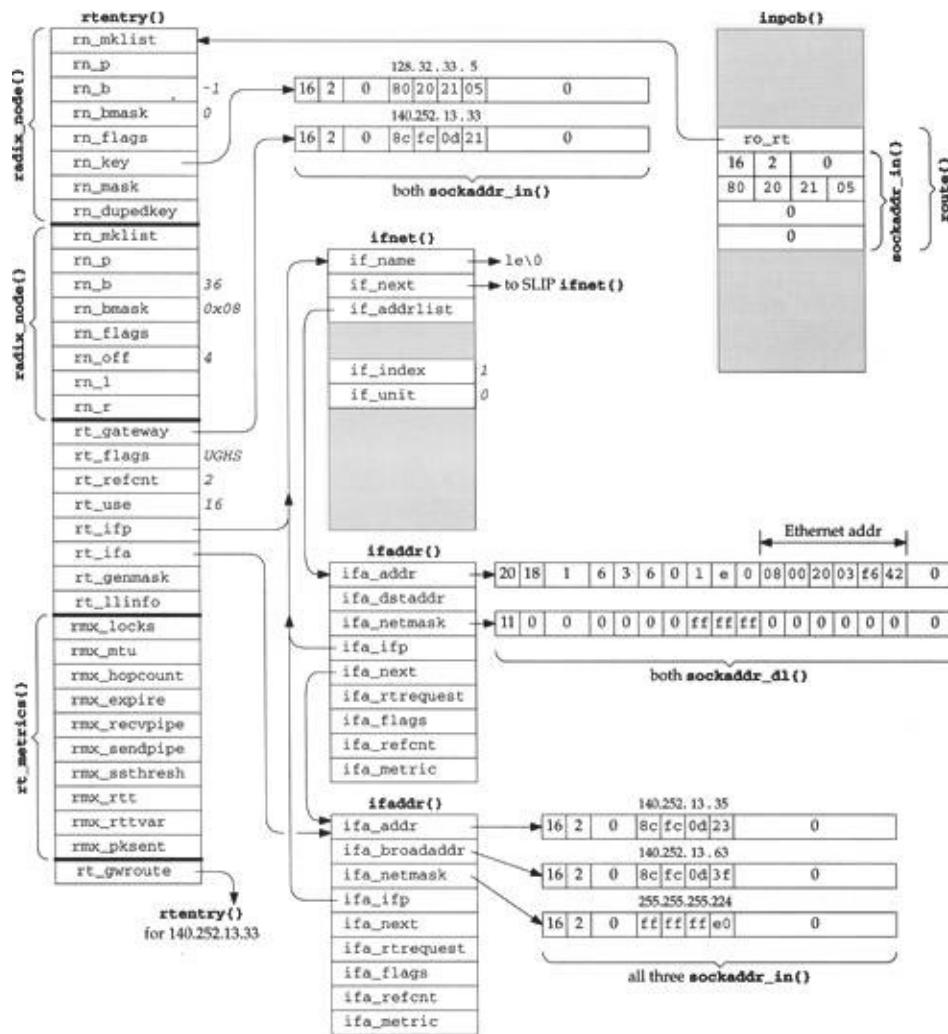
Member	OSI value	Internet value	Routing value	Unix value	XNS value	Comment
dom_family	<code>AF_ISO</code>	<code>AF_INET</code>	<code>PF_ROUTE</code>	<code>AF_UNIX</code>	<code>AF_NS</code>	
dom_init	0	0	<code>route_init</code>	0	0	
dom_rtattach	<code>rn_inithead</code>	<code>rn_inithead</code>	0	0	<code>rn_inithead</code>	
dom_rtoffset	48	32	0	0	16	in bits
dom_maxrtkey	32	16	0	0	16	in bytes

The PF\_ROUTE domain is the only one with an initialization function. Also, only the domains that require a routing table have a dom\_rtattach function, and it is always rn\_inithead. The routing domain and the Unix domain protocols do not require a routing table.

The dom\_rtoffset member is the offset, in bits, (from the beginning of the domain's socket address structure) of the first bit to be examined for routing. The size of this structure in bytes is given by dom\_maxrtkey. We saw earlier in this chapter that the offset of the IP address in the sockaddr\_in structure is 32 bits. The dom\_maxrtkey member is the size in bytes of the protocol's socket address structure: 16 for sockaddr\_in.

Figure 18.29 outlines the steps involved in initializing the routing tables.

## Figure 18.28. Summary of routing structures.



## Figure 18.29. Steps involved in initialization of routing tables.

```

main()          /* kernel initialization */
{
    ...
    ifinit();
    domaininit();
    ...
}

domaininit()    /* Figure 7.15 */
{
    ...
    ADDDOMAIN(unix);
    ADDDOMAIN(route);
    ADDDOMAIN/inet;
    ADDDOMAIN(osi);
    ...
    for ( dp = all_domains ) {
        (*dp->dom_init)();
        for ( pr = all protocols for this domain )
            (*pr->pr_init)();
    }
}

raw_init()      /* pr_init() function for SOCK_RAW/PF_ROUTE protocol */
{
    initialize head of routing protocol control blocks;
}

route_init()    /* dom_init() function for PF_ROUTE domain */
{
    rn_init();
    rtable_init();
}

rn_init()
{
    for ( dp = all_domains )
        if (dp->dom_maxrtkey > max_keylen)
            max_keylen = dp->dom_maxrtkey;
    allocate and initialize rn_zeros, rn_ones, masked_key;
    rn_inithead(&mask_rnhead); /* allocate and init tree for masks */
}

rtable_init()
{
    for ( dp = all_domains )
        (*dp->dom_rtattach)(&rt_tables[dp->dom_family]);
}

rn_inithead()   /* dom_rtattach() function for all protocol families */
{
    allocate and initialize one radix_node_head structure;
}

```

The diagram illustrates the flow of control in the kernel's initialization code. It starts with the `main()` function, which calls `ifinit()` and `domaininit()`. The `domaininit()` function then calls `ADDDOMAIN(unix)`, `ADDDOMAIN(route)`, `ADDDOMAIN/inet`, and `ADDDOMAIN(osi)`. For each domain, it enters a loop where it calls `dp->dom_init()` and then iterates over all protocols for that domain, calling `pr->pr_init()` for each. This leads to the `raw_init()` function, which initializes the head of routing protocol control blocks. Following this, the `route_init()` function is called, which in turn calls `rn_init()` and `rtable_init()`. The `rn_init()` function initializes routing numbers by determining the maximum key length and allocating memory for trees. It then calls `rn_inithead()` to initialize the radix node head structure. Finally, the `rtable_init()` function iterates over all domains and calls `dp->dom_rtattach()` for each, passing the address of the `rt_tables` array corresponding to the domain's family.

`domaininit` is called once by the kernel's main function when the system is initialized. The linked list of domain structures is built by the `ADDDOMAIN` macro and the linked list is traversed, calling each domain's `dom_init` function, if

defined. As we saw in [Figure 18.27](#), the only dom\_init function is route\_init, which is shown in [Figure 18.30](#).

### **Figure 18.30. route\_init function.**

```
route.c
49 void
50 route_init()
51 {
52     rn_init(); /* initialize all zeros, all ones, mask table */
53     rtable_init((void **) rt_tables);
54 }
```

route.c

The function rn\_init, shown in [Figure 18.32](#), is called only once.

The function rtable\_init, shown in [Figure 18.31](#), is also called only once. It in turn calls all the dom\_rtattach functions, which initialize a routing table tree for that domain.

### **Figure 18.31. rtable\_init function: call each domain's dom\_rtattach function.**

```
39 void
40 rtable_init(table)
41 void **table;
42 {
43     struct domain *dom;
44     for (dom = domains; dom; dom = dom->dom_next)
45         if (dom->dom_rtattach)
46             dom->dom_rtattach(&table[dom->dom_family],
47                                dom->dom_rtoffset);
48 }
```

- route.c

- route.c

We saw in [Figure 18.27](#) that the only dom\_rtattach function is rn\_inithead, which we describe in the next section.

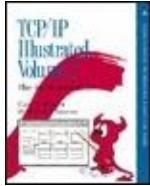
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.8 Initialization: rn\_init and rn\_inithead Functions

The function `rn_init`, shown in [Figure 18.32](#), is called once by `route_init` to initialize some of the globals used by the radix functions.

#### Figure 18.32. `rn_init` function.

---

```
750 void
751 rn_init()
752 {
753     char    *cp, *cplim;
754     struct domain *dom;
```

*- radix.c*

```

755     for (dom = domains; dom; dom = dom->dom_next)
756         if (dom->dom_maxrtkey > max_keylen)
757             max_keylen = dom->dom_maxrtkey;
758     if (max_keylen == 0) {
759         printf("rn_init: radix functions require max_keylen be set\n");
760         return;
761     }
762     R_Malloc(rn_zeros, char *, 3 * max_keylen);
763     if (rn_zeros == NULL)
764         panic("rn_init");
765     Bzero(rn_zeros, 3 * max_keylen);
766     rn_ones = cp = rn_zeros + max_keylen;
767     maskedKey = cplim = rn_ones + max_keylen;
768     while (cp < cplim)
769         *cp++ = -1;
770     if (rn_inithead((void **) &mask_rnhead, 0) == 0)
771         panic("rn_init 2");
772 }
```

- radix.c

## Determine max\_keylen

750-761

All the domain structures are examined and the global max\_keylen is set to the largest value of dom\_maxrtkey. In [Figure 18.27](#) the largest value is 32 for AF\_ISO, but in a typical system that excludes the OSI and XNS protocols, max\_keylen is 16, the size of a sockaddr\_in structure.

## Allocate and initialize rn\_zeros, rn\_ones, and maskedKey

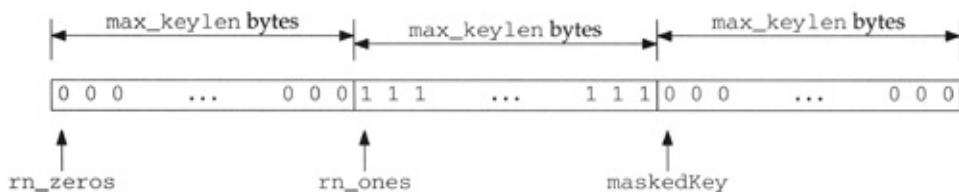
762-769

A buffer three times the size of

`max_keylen` is allocated and the pointer stored in the global `rn_zeros`. `R_Malloc` is a macro that calls the kernel's `malloc` function, specifying a type of `M_RTABLE` and `M_DONTWAIT`. We'll also encounter the macros `Bcmp`, `Bcopy`, `Bzero`, and `Free`, which call kernel functions of similar names, with the arguments appropriately type cast.

This buffer is divided into three pieces, and each piece is initialized as shown in [Figure 18.33](#).

**Figure 18.33. `rn_zeros`, `rn_ones`, and `maskedKey` arrays.**



`rn_zeros` is an array of all zero bits, `rn_ones` is an array of all one bits, and `maskedKey` is an array used to hold a temporary copy of a search key that has been masked.

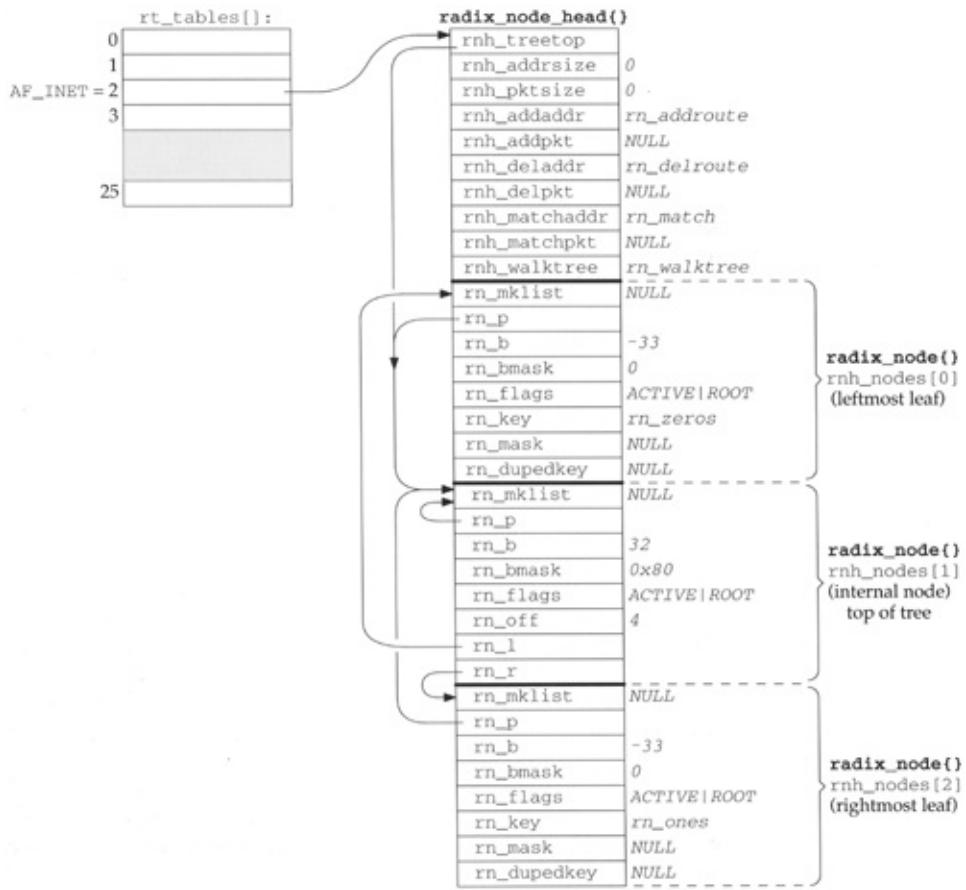
## Initialize tree of masks

770-772

The function `rn_inithead` is called to initialize the head of the routing tree for the address masks; the `radix_node_head` structure pointed to by the global `mask_rnhead` in [Figure 18.8](#).

From [Figure 18.27](#) we see that `rn_inithead` is also the `dom_attach` function for all the protocols that require a routing table. Instead of showing the source code for this function, [Figure 18.34](#) shows the `radix_node_head` structure that it builds for the Internet protocols.

**Figure 18.34. `radix_node_head` structure built by `rn_inithead` for Internet protocols.**



The three radix\_node structures form a tree: the middle of the three is the top (it is pointed to by rnh\_treetop), the first of the three is the leftmost leaf of the tree, and the last of the three is the rightmost leaf of the tree. The parent pointer of all three nodes (rn\_p) points to the middle node.

The value 32 for rnh\_nodes[1].rn\_b is the bit position to test. It is from the dom\_rtoffset member of the Internet

domain structure ([Figure 18.27](#)). Instead of performing shifts and masks during forwarding, the byte offset and corresponding byte mask are precomputed. The byte offset from the start of a socket address structure is in the rn\_off member of the radix\_node structure (4 in this case) and the byte mask is in the rn\_bmask member (0x80 in this case). These values are computed whenever a radix\_node structure is added to the tree, to speed up the comparisons during forwarding. As additional examples, the offset and byte mask for the two nodes that test bit 33 in [Figure 18.4](#) would be 4 and 0x40, respectively. The offset and byte mask for the two nodes that test bit 63 would be 7 and 0x01.

The value of -33 for the rn\_b member of both leaves is negative one minus the index of the leaf.

The key of the leftmost node is all zero bits (rn\_zeros) and the key of the rightmost node is all one bits (rn\_ones).

All three nodes have the RNF\_ROOT flag

set. (We have omitted the RNF\_ prefix.) This indicates that the node is one of the three original nodes used to build the tree. These are the only nodes with this flag.

One detail we have not mentioned is that the Network File System (NFS) also uses the routing table functions. For each mount point on the local host a radix\_node\_head structure is allocated, along with an array of pointers to these structures (indexed by the protocol family), similar to the rt\_tables array. Each time this mount point is exported, the protocol address of the host that can mount this filesystem is added to the appropriate tree for the mount point.



## Chapter 18. Radix Tree Routing Tables

---

### 18.9 Duplicate Keys and Mask Lists

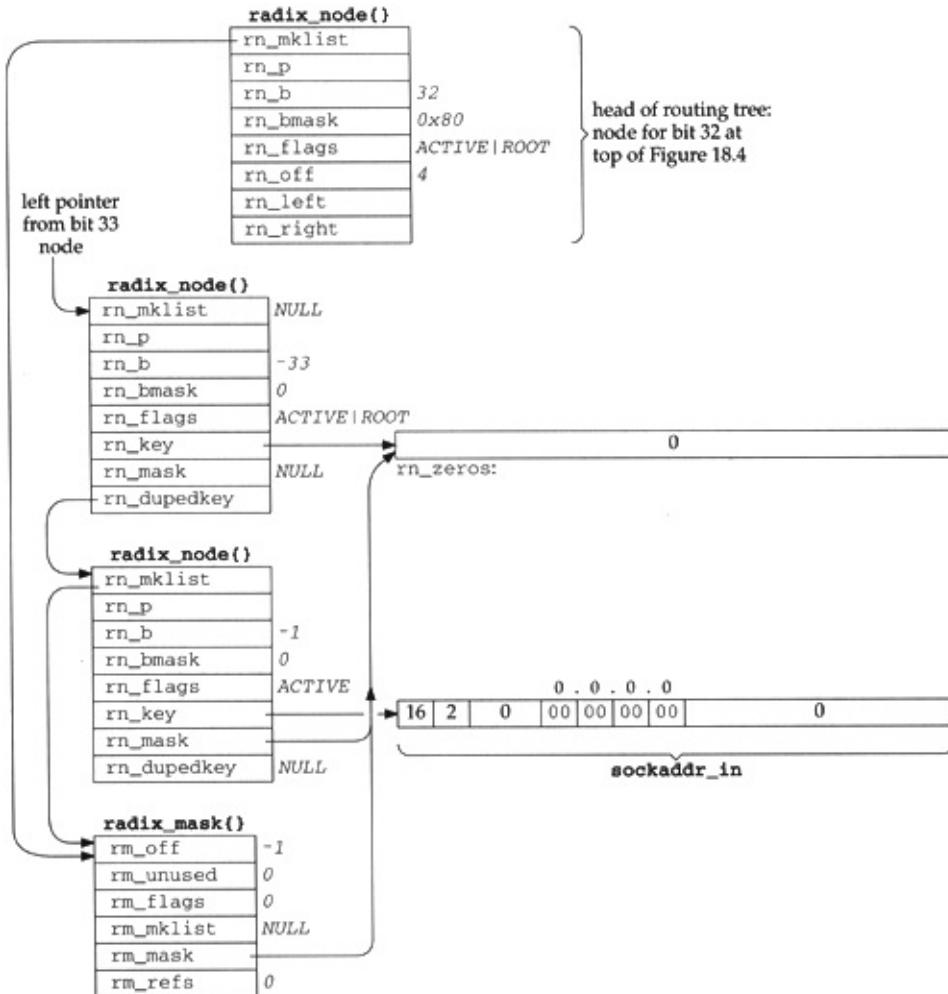
Before looking at the source code that looks up routes in the radix tree, it is important to understand two fields in the `radix_node` structure. The first is a pointer to a linked list of additional `radix_node` structures called `rnh_mklist`, which starts a linked list of `radix_masks`. These masks are used to determine if a route is a default route or an end route.

We first return to [Figure 18.4](#) and the two boxes labeled "end" and "default." These are duplicate keys. In fact, the "end" key has the RNF\_ROOT flag set (`rnh_nodes[0]` in [Figure 18.4](#)). This means that this is the same key as the default route. We will see that the rightmost end node in the tree, which has a mask of 0, was created for 255.255.255.255, but this is the wrong mask for a default route. It doesn't appear in the routing table. In general, we do not allow any key to be duplicated, if each occurrence of a key has a different mask.

[Figure 18.35](#) shows the two nodes with a duplicate key. In the figure we have removed the RNF\_ prefix for the key values.

left, and right pointers, which add nothing to the structure.

**Figure 18.35. Duplicated nodes will be merged.**



The top node is the top of the routing tree—the head of the tree. The next two nodes are leaves (their rn\_dupedkey member of the first pointing to the second). The leaves are merged into a single node, whose rn\_dupedkey member points to the rnh\_nodes[0] structure from Figure 18.4. The marker of the tree is its RNF\_ROOT flag is set. Its rn\_left and rn\_right pointers, which add nothing to the structure.

`rn_inithead` to `rn_zeros`.

The second of these leaves is the entry for the `sockaddr_in` with the value 0.0.0.0, and it has a pointer to `rn_zeros`, since equivalent masks in the tree are shared.

Normally keys are not shared, let alone share of the two end markers (those with the `RNF_IS_SHARED` flag) that are built by `rn_inithead` ([Figure 18.34](#)). The key of the left end marker is `rn_zeros` and the key of the right end marker is 0.0.0.0.

The final structure is a `radix_mask` structure associated with each node of the tree and the leaf for the default route. When the tree is used with the backtracking algorithm, it maintains a list of network mask. The list of `radix_mask` structure contains the masks that apply to subtrees starting at the specified keys, a mask list also appears with the leaves, for example:

We now show a duplicate key that is added to the resulting mask list. In [Figure 18.4](#) we have a default route for 127.0.0.0. The default mask is 0xff000000, as we show in the figure. If we divide the network ID into a 16-bit subnet ID and an 8-bit subnet ID, we can create a subnet 127.0.0 with a mask of 0xfffffff00:

```
bsdi $ route add 127.0.0.0 -netmask 0xfffffff00
```

Although it makes little practical sense to use  $n$  interest is in the resulting routing table structure common with the Internet protocols (other than default route), duplicate keys are required to provide network.

There is an implied priority in these three entries. If the search key is 127.0.0.1 it matches all three entries because it is the *most specific*: its mask (0xffffffff) covers all four bits. If the search key is 127.0.0.2 it matches both networks 127.0.0.0 and 127.0.0.1, with a mask of 0xfffffff00, is more specific than 0xffffffff. The search key 127.1.2.3 matches only the network 127.1.2.3 with a mask of 0xff000000.

[Figure 18.36](#) shows the resulting tree structure for the four entries shown in bit 33 from [Figure 18.4](#). We show two boxes for the entry 127.0.0.0 since there are two leaves with this prefix.

**Figure 18.36. Routing tree showing duplicate keys**

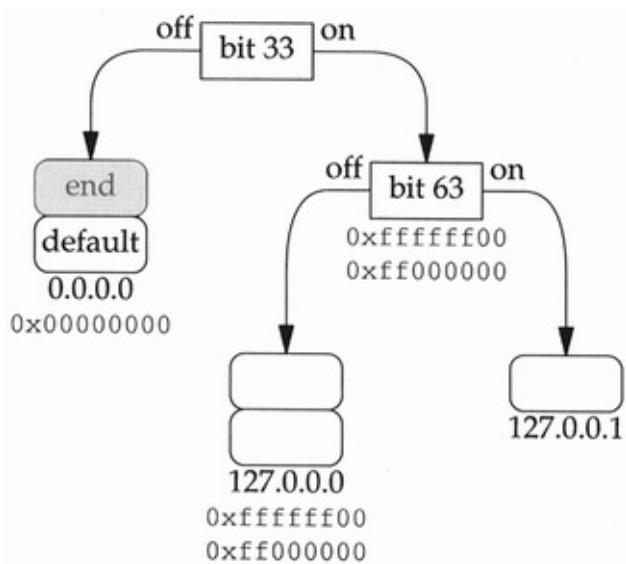
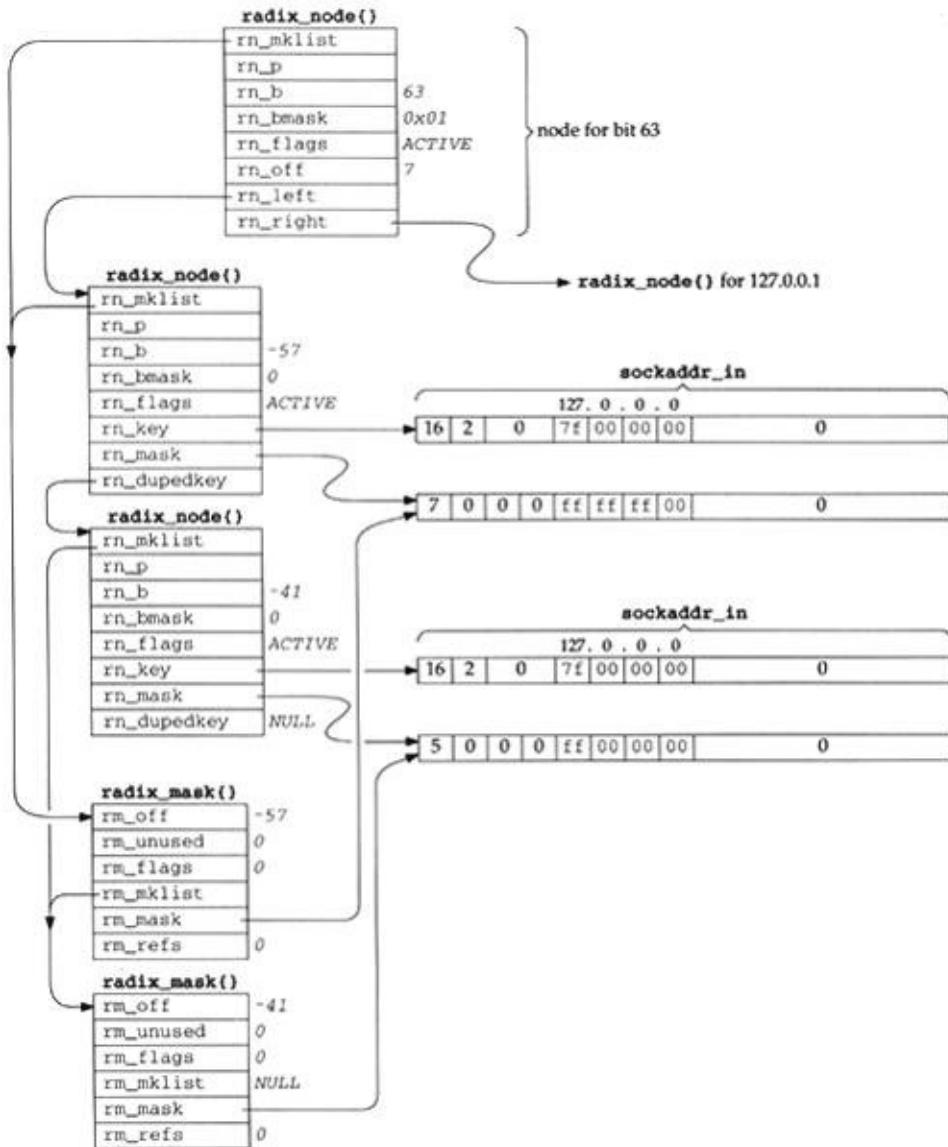


Figure 18.37 shows the resulting radix\_node ar

**Figure 18.37. Example routing table structure for network 127.0.**

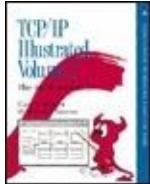


First look at the linked list of radix\_mask struct list for the top node (bit 63) consists of the ent 0xff000000. The more-specific mask comes firs The mask list for the second radix\_node (the o as that of the first. But the list for the third rad with a mask of 0xff000000.

Notice that masks with the same value are sha

are not. This is because the masks are maintained explicitly to be shared, because equal masks are used (a network route has the same mask of 0xffffffff00).

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.10 rn\_match Function

We now show the rn\_match function, which is called as the rnh\_matchaddr function for the Internet protocols. We'll see that it is called by the rtalloc1 function, which is called by the rtalloc function. The algorithm is as follows:

- 1. Start at the top of the tree and go to the leaf corresponding to the bits in the search key. Check the leaf for an exact match ([Figure 18.38](#)).**

**Figure 18.38. rn\_match function: go down tree, check for exact host**

## match.

```
135 struct radix_node *
136 rn_match(v_arg, head)
137 void *v_arg;
138 struct radix_node_head *head;
139 {
140     caddr_t v = v_arg;
141     struct radix_node *t = head->rnh_treetop, *x;
142     caddr_t cp = v, cp2, cp3;
143     caddr_t cplim, mstart;
144     struct radix_node *saved_t, *top = t;
145     int off = t->rn_off, vlen = *(u_char *) cp, matched_off;
146
147     /*
148      * Open code rn_search(v, top) to avoid overhead of extra
149      * subroutine call.
150      */
151     for (; t->rn_b >= 0;) {
152         if (t->rn_bmask & cp[t->rn_off])
153             t = t->rn_r;           /* right if bit on */
154         else
155             t = t->rn_l;           /* left if bit off */
156     }
157
158     /*
159      * See if we match exactly as a host destination
160      */
161     cp += off;
162     cp2 = t->rn_key + off;
163     cplim = v + vlen;
164     for (; cp < cplim; cp++, cp2++)
165         if (*cp != *cp2)
166             goto on1;
167
168     /*
169      * This extra grot is in case we are explicitly asked
170      * to look up the default.  Ugh!
171      */
172     if ((t->rn_flags & RNF_ROOT) && t->rn_duplicatedkey)
173         t = t->rn_duplicatedkey;
174     return t;
175 on1:
```

radix.c

- Check the leaf for a network match (Figure 18.40).
- Backtrack (Figure 18.43).

Figure 18.38 shows the first part of rn\_match.

135-145

The first argument `v_arg` is a pointer to a socket address structure, and the second argument `head` is a pointer to the `radix_node_head` structure for the protocol. All protocols call this function ([Figure 18.17](#)) but each calls it with a different `head` argument.

In the assignment statements, `off` is the `rn_off` member of the top node of the tree (4 for Internet addresses, from [Figure 18.34](#)), and `vlen` is the length field from the socket address structure of the search key (16 for Internet addresses).

## Go down the tree to the corresponding leaf

146-155

This loop starts at the top of the tree and moves down the left and right branches until a leaf is encountered (`rn_b` is less than 0). Each test of the appropriate bit is made using the precomputed byte mask in

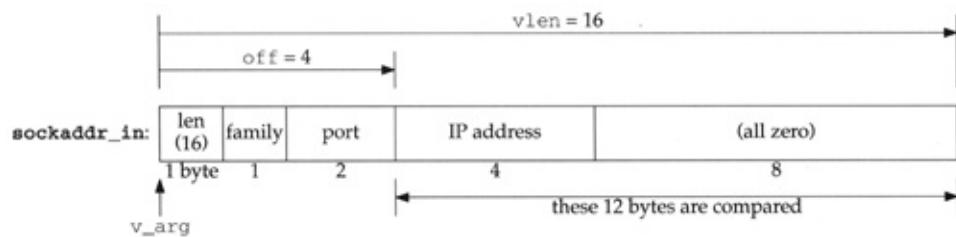
`rn_bmask` and the corresponding precomputed offset in `rn_off`. For Internet addresses, `rn_off` will be 4, 5, 6, or 7.

## Check for exact match

156-164

When the leaf is encountered, a check is first made for an exact match. *All* bytes of the socket address structure, starting at the `rn_off` value for the protocol family, are compared. This is shown in [Figure 18.39](#) for an Internet socket address structure.

**Figure 18.39. Variables during comparison of `sockaddr_in` structures.**



As soon as a mismatch is found, a jump is made to on1.

Normally the final 8 bytes of the `sockaddr_in` are 0 but proxy ARP ([Section 21.12](#)) sets one of these bytes nonzero. This allows two routing table entries for a given IP address: one for the normal IP address (with the final 8 bytes of 0) and a proxy ARP entry for the same IP address (with one of the final 8 bytes nonzero).

The length byte in [Figure 18.39](#) was assigned to `vlen` at the beginning of the function, and we'll see that `rtalloc1` uses the `family` member to select the routing table to search. The `port` is never used by the routing functions.

## Explicit check for default

165-172

[Figure 18.35](#) showed that the default route is stored as a duplicate leaf with a key of 0. The first of the duplicate leaves has the `RNF_ROOT` flag set. Hence if the `RNF_ROOT` flag is set in the matching node and the leaf contains a duplicate key the

value of the pointer rn\_dupedkey is returned (i.e., the pointer to the node containing the default route in [Figure 18.35](#)). If a default route has not been entered and the search matches the left end marker (a key of all zero bits), or if the search encounters the right end marker (a key of all one bits), the returned pointer t points to a node with the RNP\_ROOT flag set. We'll see that rtalloc1 explicitly checks whether the matching node has this flag set, and considers such a match an error.

At this point in rn\_match a leaf has been reached but it is not an exact match with the search key. The next part of the function, shown in [Figure 18.40](#), checks whether the leaf is a network match.

### **Figure 18.40. rn\_match function: check for network match.**

```

173     matched_off = cp - v;
174     saved_t = t;
175     do {
176         if (t->rn_mask) {
177             /*
178              * Even if we don't match exactly as a host;
179              * we may match if the leaf we wound up at is
180              * a route to a net.
181             */
182             cp3 = matched_off + t->rn_mask;
183             cp2 = matched_off + t->rn_key;
184             for (; cp < cplim; cp++)
185                 if ((*cp2++ ^ *cp) & *cp3++)
186                     break;
187             if (cp == cplim)
188                 return t;
189             cp = matched_off + v;
190         }
191     } while (t = t->rn_dupedkey);
192     t = saved_t;

```

---

- radix.c

## 173-174

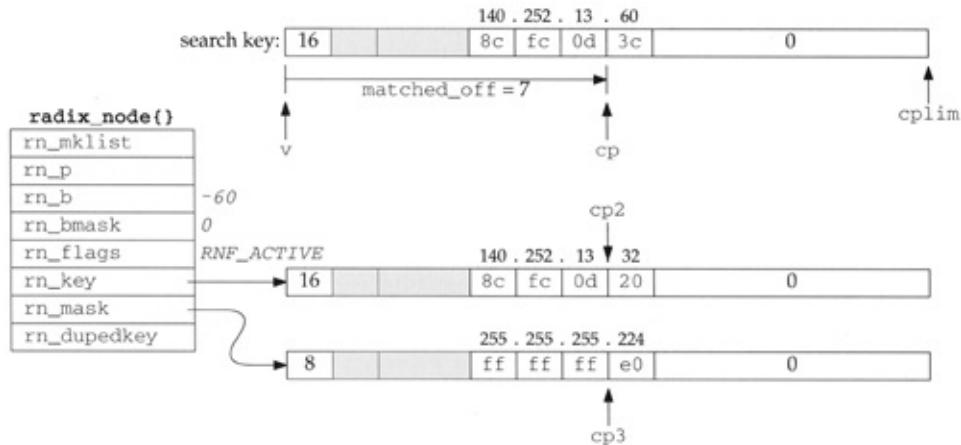
cp points to the unequal byte in the search key. matched\_off is set to the offset of this byte from the start of the socket address structure.

## 175-183

The do while loop iterates through all duplicate leaves and each one with a network mask is compared. Let's work through the code with an example. Assume we're looking up the IP address 140.252.13.60 in the routing table in [Figure 18.4](#). The search will end up at the node labeled 140.252.13.32 (bits 62 and 63 are both off), which contains a network mask. [Figure 18.41](#) shows the structures

when the for loop in Figure 18.40 starts executing.

**Figure 18.41. Example for network mask comparison.**



The search key and the routing table key are both `sockaddr_in` structures, but the length of the mask is different. The mask length is the minimum number of bytes containing nonzero values. All the bytes past this point, up through `max_keylen`, are 0.

184-190

The search key is exclusive ORed with the routing table key, and the result logically

ANDed with the network mask, one byte at a time. If the resulting byte is ever nonzero, the loop terminates because they don't match ([Exercise 18.1](#)). If the loop terminates normally, however, the search key ANDed with the network mask matches the routing table entry. The pointer to the routing table entry is returned.

[Figure 18.42](#) shows how this example matches, and how the IP address 140.252.13.188 does not match, looking at just the fourth byte of the IP address. The search for both IP addresses ends up at this node since both addresses have bits 57, 62, and 63 off.

### Figure 18.42. Example of search key match using network mask.

	search key = 140.252.13.60	search key = 140.252.13.188
search key byte (*cp):	0011 1100 = 3c	1011 1100 = bc
routing table key byte (*cp2):	0010 0000 = 20	0010 0000 = 20
exclusive OR:	0001 1100	1001 1100
network mask byte (*cp3):	1110 0000 = e0	1110 0000 = e0
logical AND:	0000 0000	1000 0000

The first example (140.252.13.60)

matches since the result of the logical AND is 0 (and all the remaining bytes in the address, the key, and the mask are all 0). The other example does not match since the result of the logical AND is nonzero.

191

If the routing table entry has duplicate keys, the loop is repeated for each key.

The final portion of rn\_match, shown in [Figure 18.43](#), backtracks up the tree, looking for a network match or a match with the default.

**Figure 18.43. rn\_match function:  
backtrack up the tree.**

---

```

193     /* start searching up the tree */
194     do {
195         struct radix_mask *m;
196         t = t->rn_p;
197         if (m = t->rn_mklist) {
198             /*
199              * After doing measurements here, it may
200              * turn out to be faster to open code
201              * rn_search_m here instead of always
202              * copying and masking.
203              */
204             off = min(t->rn_off, matched_off);
205             mstart = maskedKey + off;
206             do {
207                 cp2 = mstart;
208                 cp3 = m->rm_mask + off;
209                 for (cp = v + off; cp < cplim;)
210                     *cp2++ = *cp++ & *cp3++;
211                 x = rn_search(maskedKey, t);
212                 while (x && x->rn_mask != m->rm_mask)
213                     x = x->rn_duplicatedkey;
214                 if (x &&
215                     (Bcmp(mstart, x->rn_key + off,
216                           vlen - off) == 0))
217                     return x;
218             } while (m = m->rm_mklist);
219         }
220     } while (t != top);
221     return 0;
222 };

```

---

- radix.c

## 193-195

The do while loop continues up the tree, checking each level, until the top has been checked.

## 196

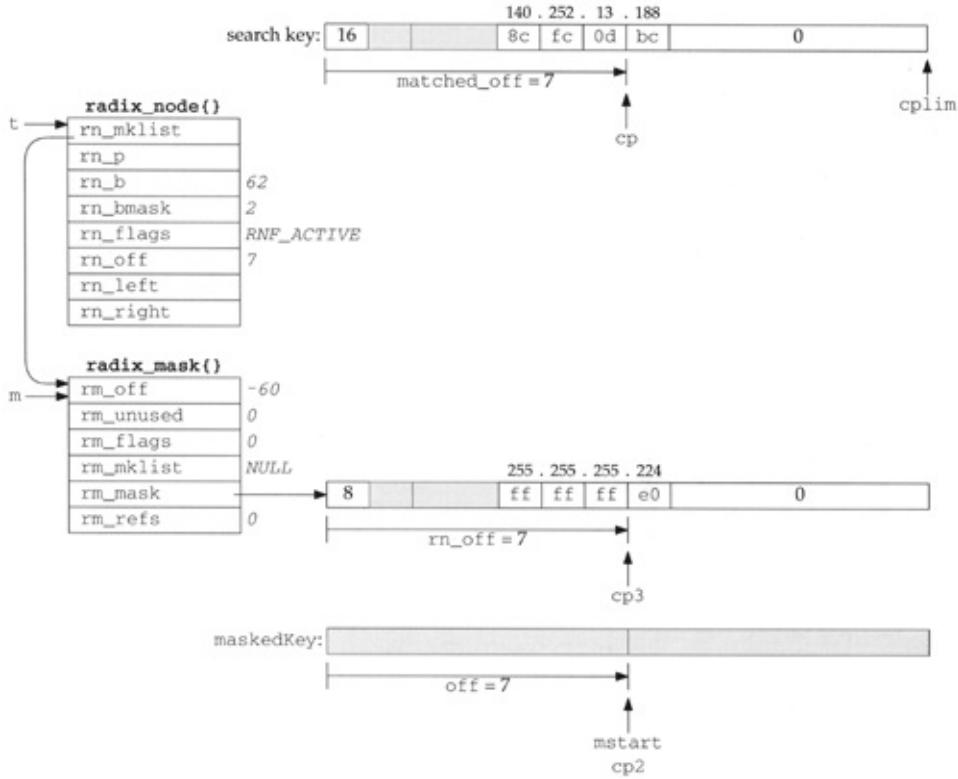
The pointer t is replaced with the pointer to the parent node, moving up one level. Having the parent pointer in each node simplifies backtracking.

## 197-210

Each level is checked only if the internal node has a nonnull list of masks. rn\_mklist is a pointer to a linked list of radix\_mask structures, each containing a mask that applies to the subtree starting at that node. The inner do while loop iterates through each radix\_mask structure on the list.

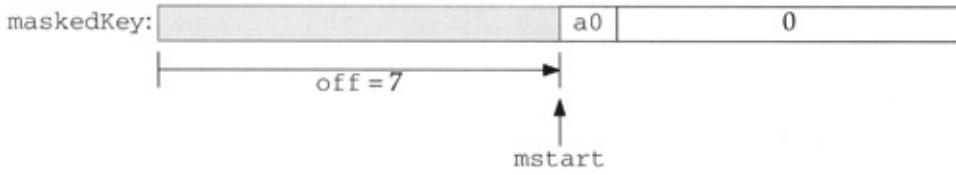
Using the previous example, 140.252.13.188, [Figure 18.44](#) shows the various data structures when the innermost for loop starts. This loop logically ANDs each byte of the search key with each byte of the mask, storing the result in the global maskedKey. The mask value is 0xfffffe0 and the search would have backtracked from the leaf for 140.252.13.32 in [Figure 18.4](#) two levels to the node that tests bit 62.

**Figure 18.44. Preparation to search again using masked search key.**



Once the for loop completes, the masking is complete, and `rn_search` (shown in [Figure 18.48](#)) is called with `maskedKey` as the search key and the pointer `t` as the top of the subtree to search. [Figure 18.45](#) shows the value of `maskedKey` for our example.

**Figure 18.45. `maskedKey` when `rn_search` is called.**

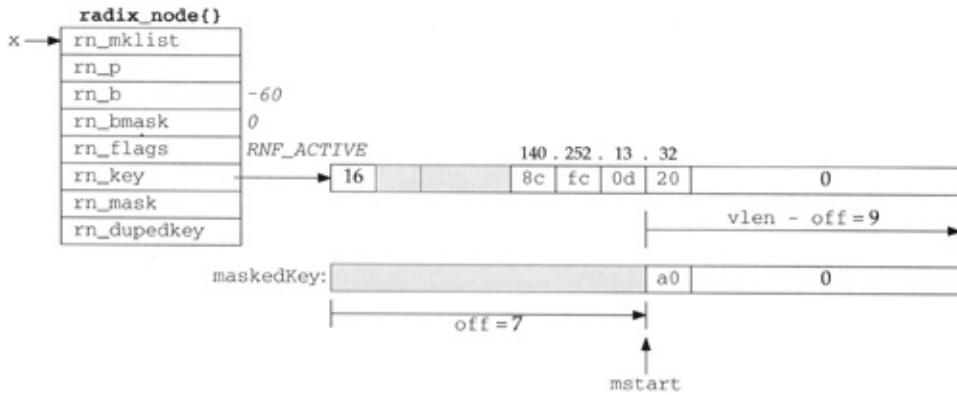


The byte 0xa0 is the logical AND of 0xbc (188, the search key) and 0xe0 (the mask).

211

rn\_search proceeds down the tree from its starting point, branching right or left depending on the key, until a leaf is reached. In this example the search key is the 9 bytes shown in [Figure 18.45](#) and the leaf that's reached is the one labeled 140.252.13.32 in [Figure 18.4](#), since bits 62 and 63 are off in the byte 0xa0. [Figure 18.46](#) shows the data structures when Bcmp is called to check if a match has been found.

**Figure 18.46. Comparison of maskedKey and new leaf.**



Since the 9-byte strings are not the same, the comparison fails.

212-221

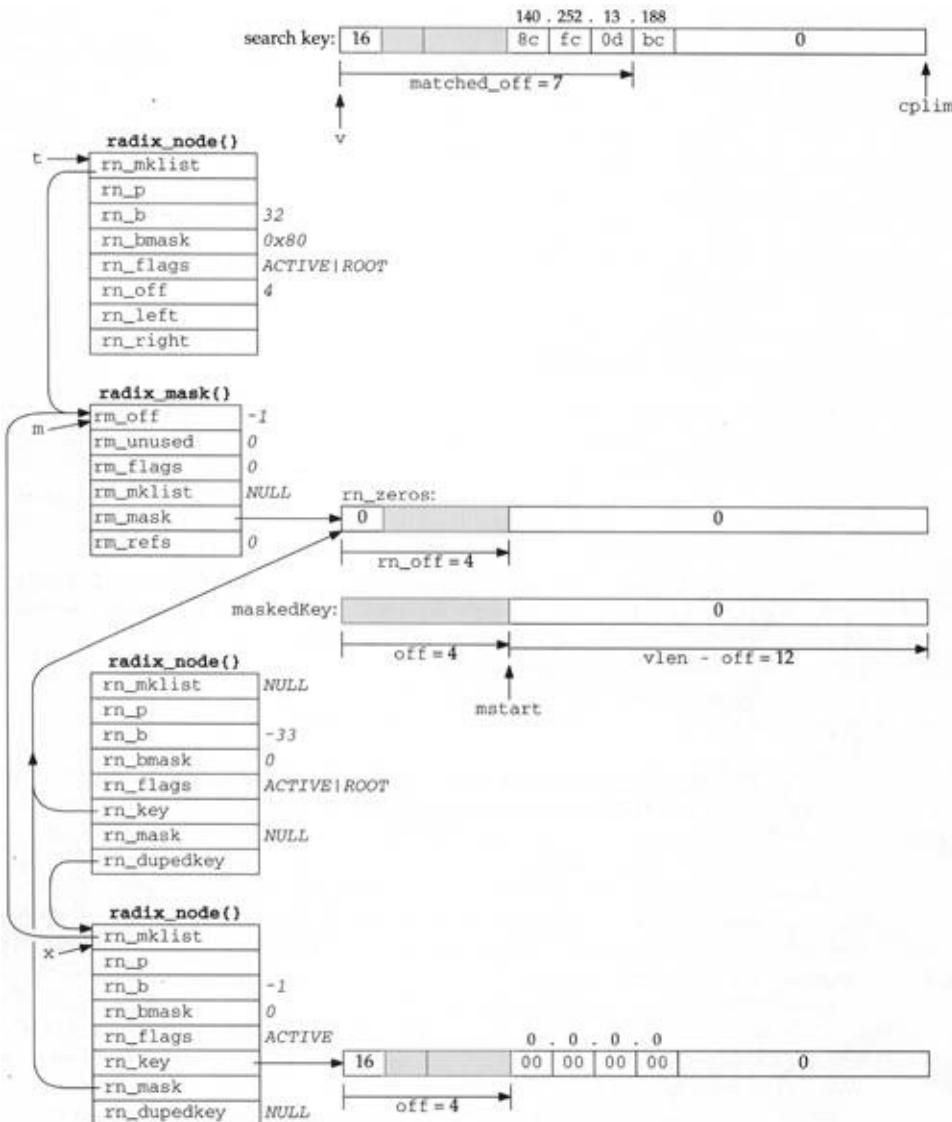
This while loop handles duplicate keys, each with a different mask. The only key of the duplicates that is compared is the one whose rn\_mask pointer equals m->rm\_mask. As an example, recall [Figures 18.36](#) and [18.37](#). If the search starts at the node for bit 63, the first time through the inner do while loop m points to the radix\_mask structure for 0xffffffff00. When rn\_search returns the pointer to the first of the duplicate leaves for 127.0.0.0, the rm\_mask of this leaf equals m->rm\_mask, so Bcmp is called. If the comparison fails, m is replaced with the pointer to the next radix\_mask structure on the list (the one with a mask of 0xff000000) and the do

while loop iterates around again with the new mask. rn\_search again returns the pointer to the first of the duplicate leaves for 127.0.0.0, but its rn\_mask does not equal m->rm\_rnask. The while steps to the next of the duplicate leaves and its rn\_mask is the right one.

Returning to our example with the search key of 140.252.13.188, since the search from the node that tests bit 62 failed, the backtracking continues up the tree until the top is reached, which is the next node up the tree with a nonnull rn\_mklist.

[Figure 18.47](#) shows the data structures when the top node of the tree is reached. At this point maskedKey is computed (it is all zero bits) and rn\_search starts at this node (the top of the tree) and continues down the two left branches to the leaf labeled "default" in [Figure 18.4](#).

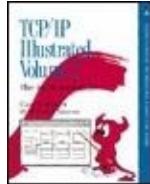
**Figure 18.47. Backtrack to top of tree and rn\_search that locates default leaf.**



When rn\_search returns, x points to the radix\_node with an rn\_b of -33, which is the first leaf encountered after the two left branches from the top of the tree. But x->rn\_mask (which is null) does not equal m->rm\_mask, so x is replaced with x->rn\_dupedkey. The test of the while loop occurs again, but now x->rn\_mask equals

`m->rm_mask`, so the while loop terminates. Bcmp compares the 12 bytes of 0 starting at `mstart` with the 12 bytes of 0 stating at `x->rn_key` plus 4, and since they're equal, the function returns the pointer `x`, which points to the entry for the default route.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.11 rn\_search Function

rn\_search was called in the previous section from rn\_match to search a subtree of the routing table.

#### Figure 18.48. rn\_search function.

```
79 struct radix_node *
80 rn_search(v_arg, head)
81 void *v_arg;
82 struct radix_node *head;
83 {
84     struct radix_node *x;
85     caddr_t v;
86     for (x = head, v = v_arg; x->rn_b >= 0;) {
87         if (x->rn_bmask & v[x->rn_off])
88             x = x->rn_r; /* right if bit on */
89         else
90             x = x->rn_l; /* left if bit off */
91     }
92     return (x);
93 }
```

-radix.c

This loop is similar to the one in [Figure 18.38](#). It compares one bit in the search key at each node, branching left if the bit is off or right if the bit is on, terminating when a leaf is encountered. The pointer to that leaf is returned.

---

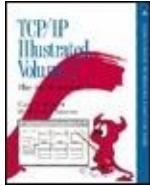
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 18. Radix Tree Routing Tables

### 18.12 Summary

Each routing table entry is identified by a key: the destination IP address in the case of the Internet protocols, which is either a host address or a network address with an associated network mask. Once the entry is located by searching for the key, additional information in the entry specifies the IP address of a router to which datagrams should be sent for the destination, a pointer to the interface to use, metrics, and so on.

The information maintained by the Internet protocols is the route structure, composed of just two elements: a pointer

to a routing table entry and the destination address. We'll encounter one of these route structures in each of the Internet protocol control blocks used by UDP, TCP, and raw IP.

The Patricia tree data structure is well suited to routing tables. Routing table lookups occur much more frequently than adding or deleting routes, so from a performance standpoint using Patricia trees for the routing table makes sense. Patricia trees provide fast lookups at the expense of additional work in adding and deleting. Measurements in [[Sklower 1991](#)] comparing the radix tree approach to the Net/1 hash table show that the radix tree method is about two times faster in building a test tree and four times faster in searching.

## Exercises

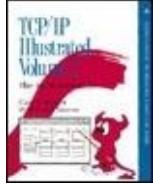
We said with [Figure 18.3](#) that the general condition for matching a routing table entry is that the search

**18.1** key logically ANDed with the routing table mask equal the routing table key. But in [Figure 18.40](#) a different test is used. Build a logic truth table showing that the two tests are the same.

**18.2** Assume a Net/3 system needs a routing table with 20,000 entries (IP addresses). Approximately how much memory is required for this, ignoring the space required for the masks?

**18.3** What is the limit imposed on the length of a routing table key by the radix\_node structure?





**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 19. Routing Requests and Routing Messages

[Section 19.1. Introduction](#)

[Section 19.2. rtalloc and rtalloc1 Functions](#)

[Section 19.3. RTFREE Macro and rtfree Function](#)

[Section 19.4. rtrequest Function](#)

[Section 19.5. rt\\_setgate Function](#)

[Section 19.6. rtinit Function](#)

[Section 19.7. rtredirect Function](#)

[Section 19.8. Routing Message Structures](#)

[Section 19.9. rt\\_missmsg Function](#)

[Section 19.10. rt\\_ifmsg Function](#)

## [Section 19.11. rt\\_newaddrmsg Function](#)

## [Section 19.12. rt\\_msg1 Function](#)

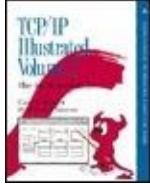
## [Section 19.13. rt\\_msg2 Function](#)

## [Section 19.14. sysctl\\_rtable Function](#)

## [Section 19.15. sysctl\\_dumpentry Function](#)

## [Section 19.16. sysctl\\_iflist Function](#)

## [Section 19.17. Summary](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.1 Introduction

The various protocols within the kernel don't access the routing trees directly, using the functions from the previous chapter, but instead call a few functions that we describe in this chapter: rtalloc and rtalloc1 are two that perform routing table lookups, rtrequest adds and deletes routing table entries, and rtinit is called by most interfaces when the interface goes up or down.

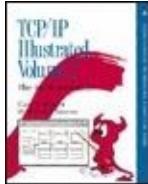
Routing messages communicate information in two directions. A process such as the route command or one of the routing daemons (routed or gated) writes

routing messages to a routing socket, causing the kernel to add a new route, delete an existing route, or modify an existing route. The kernel also generates routing messages that can be read by any routing socket when events occur in which the processes might be interested: an interface has gone down, a redirect has been received, and so on. In this chapter we cover the formats of these routing messages and the information contained therein, and we save our discussion of routing sockets until the next chapter.

Another interface provided by the kernel to the routing tables is through the `sysctl` system call, which we describe at the end of this chapter. This system call allows a process to read the entire routing table or a list of all the configured interfaces and interface addresses.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.2 rtalloc and rtalloc1 Functions

rtalloc and rtalloc1 are the functions normally called to look up an entry in the routing table. [Figure 19.1](#) shows rtalloc.

**Figure 19.1. rtalloc function.**

```
route.c
58 void
59 rtalloc(ro)
60 struct route *ro;
61 {
62     if (ro->ro_rt && ro->ro_rt->rt_ifp && (ro->ro_rt->rt_flags & RTF_UP))
63         return; /* XXX */
64     ro->ro_rt = rtalloc1(&ro->ro_dst, 1);
65 }
```

route.c

The argument `ro` is often the pointer to a route structure contained in an Internet PCB ([Chapter 22](#)) which is used by UDP and TCP. If `ro` already points to an `rtentry` structure (`ro_rt` is nonnull), and that structure points to an interface structure, and the route is up, the function returns. Otherwise `rtalloc1` is called with a second argument of 1. We'll see the purpose of this argument shortly.

`rtalloc1`, shown in [Figure 19.2](#), calls the `rnh_matchaddr` function, which is always `rn_match` ([Figure 18.17](#)) for Internet addresses.

## Figure 19.2. `rtalloc1` function.

---

```

66 struct rtentry *
67 rtalloc1(dst, report)
68 struct sockaddr *dst;
69 int      report;
70 {
71     struct radix_node_head *rnh = rt_tables[dst->sa_family];
72     struct rtentry *rt;
73     struct radix_node *rn;
74     struct rtentry *newrt = 0;
75     struct rt_addrinfo info;
76     int      s = splnet(), err = 0, msgtype = RTM_MISS;
77
78     if (rnh && (rn = rnh->rn_matchaddr((caddr_t) dst, rnh)) &&
79         ((rn->rn_flags & RNF_ROOT) == 0)) {
80         newrt = rt = (struct rtentry *) rn;
81         if (report && (rt->rt_flags & RTF_CLONING)) {
82             err = rtrequest(RTM_RESOLVE, dst, SA(0),
83                             SA(0), 0, &newrt);
84             if (err) {
85                 newrt = rt;
86                 rt->rt_refcnt++;
87                 goto miss;
88             }
89             if ((rt = newrt) && (rt->rt_flags & RTF_XRESOLVE)) {
90                 msgtype = RTM_RESOLVE;
91                 goto miss;
92             }
93             rt->rt_refcnt++;
94         } else {
95             rtstat.rts_unreach++;
96             miss:if (report) {
97                 bzero((caddr_t) &info, sizeof(info));
98                 info.rti_info[RTAX_DST] = dst;
99                 rt_missmsg(msgtype, &info, 0, err);
100            }
101        }
102        splx(s);
103        return (newrt);
104    }

```

---

route.c

## 66-76

The first argument is a pointer to a socket address structure containing the address to search for. The `sa_family` member selects the routing table to search.

## Call `rn_match`

## 77-78

If the following three conditions are met, the search is successful.

## **1. A routing table exists for the protocol family,**

- rn\_match returns a nonnull pointer, and
- the matching radix\_node does not have the RNF\_ROOT flag set.

Remember that the two leaves that mark the end of the tree both have the RNF\_ROOT flag set.

## **Search fails**

### **94-101**

If the search fails because any one of the three conditions is not met, the statistic rts\_unreach is incremented and if the second argument to rtalloc1 (report) is nonzero, a routing message is generated that can be read by any interested processes on a routing socket. The routing message has the type RTM\_MISS, and the function returns a null pointer.

79

If all three of the conditions are met, the lookup succeeded and the pointer to the matching radix\_node is stored in rt and newrt. Notice that in the definition of the rtentry structure ([Figure 18.24](#)) the two radix\_node structures are at the beginning, and, as shown in [Figure 18.8](#), the first of these two structures contains the leaf node. Therefore the pointer to a radix\_node structure returned by rn\_match is really a pointer to an rtentry structure, which is the matching leaf node.

## Create clone entries

80-82

If the caller specified a nonzero second argument, and if the RTF\_CLONING flag is set, rtrequest is called with a command of RTM\_RESOLVE to create a new rtentry structure that is a clone of the one that was located. This feature is used by ARP and for multicast addresses.

## **Clone creation fails**

83-87

If rtrequest returns an error, newrt is set back to the entry returned by rn\_match and its reference count is incremented. A jump is made to miss where an RTM\_MISS message is generated.

## **Check for external resolution**

88-91

If rtrequest succeeds but the newly cloned entry has the RTF\_XRESOLVE flag set, a jump is made to miss, this time to generate an RTM\_RESOLVE message. The intent of this message is to notify a user process when the route is created, and it could be used with the conversion of IP addresses to X.121 addresses.

## **Increment reference count for normal successful search**

## 92-93

When the search succeeds but the RTF\_CLONING flag is not set, this statement increments the entry's reference count. This is the normal flow through the function, which then returns the nonnull pointer.

For a small function, rtalloc1 has many options in how it operates. There are seven different flows through the function, summarized in [Figure 19.3](#).

**Figure 19.3. Summary of operation of rtalloc1.**

	report argument	RTF_-CLONING flag	RTM_-RESOLVE return	RTF_-XRESOLVE flag	routing message generated	rt_refcnt	return value
entry not found	0						null
	1				RTM_MISS		null
entry found		0				++	ptr
	0					++	ptr
	1	1	OK	0		++	ptr
	1	1	OK	1	RTM_RESOLVE	++	ptr
	1	1	error		RTM_MISS	++	ptr

We note that the first two rows (entry not found) are impossible if a default route exists. Also we show rt\_refcnt being incremented in the fifth and sixth rows

when the call to rtrequest with a command of RTM\_RESOLVE is OK. The increment is done by rtrequest.

---

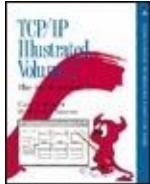
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.3 RTFREE Macro and rtfree Function

The RTFREE macro, shown in Figure 19.4, calls the rtfree function only if the reference count is less than or equal to 1, otherwise it just decrements the reference count.

**Figure 19.4. RTFREE macro.**

```
209 #define RTFREE(rt) \
210     if ((rt)->rt_refcnt <= 1) \
211         rtfree(rt); \
212     else \
213         (rt)->rt_refcnt--; /* no need for function call */
```

The rtfree function, shown in [Figure 19.5](#), releases an rtentry structure when there are no more references to it. We'll see in [Figure 22.7](#), for example, that when a protocol control block is released, if it points to a routing entry, rtfree is called.

**Figure 19.5. rtfree function: release an rtentry structure.**

```
route.c
105 void
106 rtfree(rt)
107 struct rtentry *rt;
108 {
109     struct ifaddr *ifa;
110
111     if (rt == 0)
112         panic("rtfree");
113     rt->rt_refcnt--;
114     if (rt->rt_refcnt <= 0 && (rt->rt_flags & RTF_UP) == 0) {
115         if (rt->rt_nodes->rn_flags & (RNF_ACTIVE | RNF_ROOT))
116             panic("rtfree 2");
117         rttrash--;
118         if (rt->rt_refcnt < 0) {
119             printf("rtfree: %x not freed (neg refs)\n", rt);
120             return;
121         }
122         ifa = rt->rt_ifa;
123         IFAFREE(ifa);
124         Free(rt_key(rt));
125         Free(rt);
126     }
```

route.c

105-115

The entry's reference count is decremented and if it is less than or equal to 0 and the route is not usable, the entry can be released. If either of the flags

RNF\_ACTIVE or RNF\_ROOT are set, this is an internal error. If RNF\_ACTIVE is set, this structure is still part of the routing table tree. If RNF\_ROOT is set, this structure is one of the end markers built by rn\_inithead.

116

rttrash is a debugging counter of the number of routing entries not in the routing tree, but not released. It is incremented by rtrequest when it begins deleting a route, and then decremented here. Its value should normally be 0.

## Release interface reference

117-122

A check is made that the reference count is not negative, and then IFAFREE decrements the reference count for the ifaddr structure and releases it by calling ifafree when it reaches 0.

## Release routing memory

123-124

The memory occupied by the routing entry key and its gateway is released. We'll see in `rt_setgate` that the memory for both is allocated in one contiguous chunk, allowing both to be released with a single call to `Free`. Finally the `rtentry` structure itself is released.

## Routing Table Reference Counts

The handling of the routing table reference count, `rt_refcnt`, differs from most other reference counts. We see in [Figure 18.2](#) that most routes have a reference count of 0, yet the routing table entries without any references are not deleted. We just saw the reason in `rtfree`: an entry with a reference count of 0 is not deleted unless the entry's `RTF_UP` flag is not set. The only time this flag is cleared is by `rtrequest` when a route is deleted from the routing tree.

Most routes are used in the following fashion.

- If the route is created automatically as a route to an interface when the interface is configured (which is typical for Ethernet interfaces, for example), then `rtinit` calls `rtrequest` with a command of `RTM_ADD`, creating the new entry and setting the reference count to 1. `rtinit` then decrements the reference count to 0 before returning.

A point-to-point interface follows a similar procedure, so the route starts with a reference count of 0.

If the route is created manually by the `route` command or by a routing daemon, a similar procedure occurs, with `route_output` calling `rtrequest` with a command of `RTM_ADD`, setting the reference count to 1. This is then decremented by `route_output` to 0 before it returns.

Therefore all newly created routes start with a reference count of 0.

- When an IP datagram is sent on a socket, be it TCP or UDP, we saw that

`ip_output` calls `rtalloc`, which calls `rtalloc1`. In [Figure 19.3](#) we saw that the reference count is incremented by `rtalloc1` if the route is found.

The located route is called a *held route*, since a pointer to the routing table entry is being held by the protocol, normally in a route structure contained within a protocol control block. An `rtentry` structure that is being held by someone else cannot be deleted, which is why `rtfree` doesn't release the structure until its reference count reaches 0.

- A protocol releases a held route by calling `RTFREE` or `rtfree`. We saw this in [Figure 8.24](#) when `ip_output` detects a change in the destination address. We'll encounter it in [Chapter 22](#) when a protocol control block that holds a route is released.

Part of the confusion we'll encounter in the code that follows is that `rtalloc1` is often called to look up a route in order to verify that a route to the destination exists, but

when the caller doesn't want to hold the route. Since rtalloc1 increments the counter, the caller immediately decrements it.

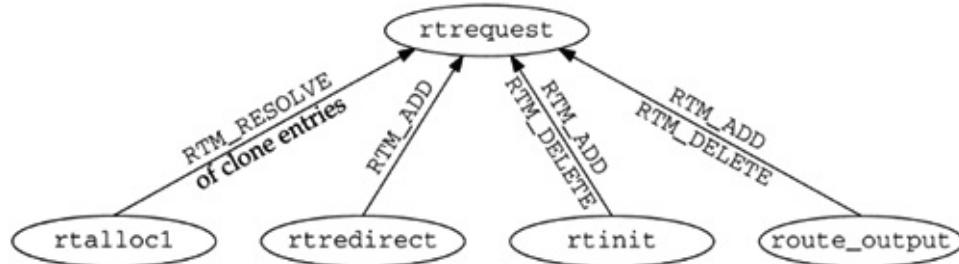
Consider a route being deleted by rtrequest. The RTF\_UP flag is cleared, and if no one is holding the route (its reference count is 0), rtfree should be called. But rtfree considers it an error for the reference count to go below 0, so rtrequest checks whether its reference count is less than or equal to 0, and, if so, increments it and calls rtfree. Normally this sets the reference count to 1 and rtfree decrements it to 0 and deletes the route.

## Chapter 19. Routing Requests and Routes

### 19.4 rtrequest Function

The `rtrequest` function is the focal point for adding and deleting routes. Figure 19.6 shows some of the other functions that can call `rtrequest`.

**Figure 19.6. Summary of function calls to rtrequest**



`rtrequest` is a switch statement with one case per message type. The first case is `RTM_RESOLVE` of clone entries. Figure 19.7 shows the start of the `rtrequest` function.

**Figure 19.7. `rtrequest` function**

---

```

290 int
291 rtrequest(req, dst, gateway, netmask, flags, ret_nrt)
292 int     req, flags;
293 struct sockaddr *dst, *gateway, *netmask;
294 struct rtentry **ret_nrt;
295 {
296     int      s = splnet();
297     int      error = 0;
298     struct rtentry *rt;
299     struct radix_node *rn;
300     struct radix_node_head *rnh;
301     struct ifaddr *ifa;
302     struct sockaddr *ndst;
303 #define senderr(x) { error = x ; goto bad; }

304     if ((rnh = rt_tables[dst->sa_family]) == 0)
305         senderr(ESRCH);
306     if (flags & RTF_HOST)
307         netmask = 0;

308     switch (req) {
309     case RTM_DELETE:
310         if ((rn = rnh->rnh_deladdr(dst, netmask, rnh)) == 0)
311             sender(ESRCH);
312         if (rn->rn_flags & (RNF_ACTIVE | RNF_ROOT))
313             panic("rtrequest delete");
314         rt = (struct rtentry *) rn;
315         rt->rt_flags &= ~RTF_UP;
316         if (rt->rt_gwroute) {
317             rt = rt->rt_gwroute;
318             RTFREE(rt);
319             (rt = (struct rtentry *) rn)->rt_gwroute = 0;
320         }
321         if ((ifa = rt->rt_ifa) && ifa->ifa_rtrequest)
322             ifa->ifa_rtrequest(RTM_DELETE, rt, SA(0));
323         rttrash++;
324         if (ret_nrt)
325             *ret_nrt = rt;
326         else if (rt->rt_refcnt <= 0) {
327             rt->rt_refcnt++;
328             rtfree(rt);
329         }
330         break;

```

---

290-307

The second argument, dst, is a socket address deleted from the routing table. The sa\_family argument indicates a host route (instead of a route), or null, ignoring any value the caller may have passed.

## Delete from routing tree

309-315

The rnh\_deladdr function (rn\_delete from [Figure 17-1](#)) traverses the tree and returns a pointer to the corresponding

## **Remove reference to gateway routing table entry**

316-320

If the entry is an indirect route through a gateway, it removes the gateway's entry and deletes it if the count is zero. It then sets rt and rt is set back to point to the entry that was

## **Call interface request function**

321-322

If an ifa\_rtrequest function is defined for this interface, it is called by ARP, for example, in [Chapter 21](#) to delete the route.

## **Return pointer or release reference**

323-330

The rttrash global is incremented because the caller does not own the route. If the caller wants the pointer to the route, it must increment its own reference count.

tree (if `ret_nrt` is nonnull), then that pointer is the caller's responsibility to call `rtfree` when it is: entry can be released: if the reference count is `rtfree` is called. The break causes the function t

[Figure 19.8](#) shows the next part of the function This function is called with this command only f from an entry with the `RTF_CLONING` flag set.

**Figure 19.8. `rtrequest` function**

```
route.c
331     case RTM_RESOLVE:
332         if (ret_nrt == 0 || (rt = *ret_nrt) == 0)
333             senderr(EINVAL);
334         ifa = rt->rt_ifa;
335         flags = rt->rt_flags & ~RTF_CLONING;
336         gateway = rt->rt_gateway;
337         if ((netmask = rt->rt_genmask) == 0)
338             flags |= RTF_HOST;
339         goto makeroute;
```

route.c

331-339

The final argument, `ret_nrt`, is used differently entry with the `RTF_CLONING` flag set ([Figure 19.8](#)). pointer, the same flags (with the `RTF_CLONING` entry being cloned has a null `rt_genmask` point because it is a host route; otherwise the new e the new entry is copied from the `rt_genmask` v network mask at the end of this section. This ca the next figure.

Figure 19.9 shows the RTM\_ADD command.

## Figure 19.9. rtrequest fun

```
route.c
340     case RTM_ADD:
341         if ((ifa = ifa_ifwithroute(flags, dst, gateway)) == 0)
342             senderr(ENETUNREACH);
343
344         makeroute:
345             R_Malloc(rt, struct rtentry *, sizeof(*rt));
346             if (rt == 0)
347                 senderr(ENOBUFS);
348             Bzero(rt, sizeof(*rt));
349             rt->rt_flags = RTF_UP | flags;
350             if (rt_setgate(rt, dst, gateway)) {
351                 Free(rt);
352                 senderr(ENOBUFS);
353             }
354             ndst = rt_key(rt);
355             if (netmask) {
356                 rt_maskedcopy(dst, ndst, netmask);
357             } else
358                 Bcopy(dst, ndst, dst->sa_len);
359             rn = rnh->rnh_addaddr((caddr_t)ndst, (caddr_t)netmask,
360                                     rnh, rt->rt_nodes);
361             if (rn == 0) {
362                 if (rt->rt_gwroute)
363                     rtfree(rt->rt_gwroute);
364                 Free(rt_key(rt));
365                 Free(rt);
366                 senderr(EEXIST);
367             }
368             ifa->if_a_refcnt++;
369             rt->rt_ifa = ifa;
370             rt->rt_ifp = ifa->if_a_ifp;
371             if (req == RTM_RESOLVE)
372                 rt->rt_rmx = (*ret_nrt)->rt_rmx; /* copy metrics */
373             if (ifa->if_a_rtrequest)
374                 ifa->if_a_rtrequest(req, rt, SA(ret_nrt ? *ret_nrt : 0));
375             if (ret_nrt) {
376                 *ret_nrt = rt;
377                 rt->rt_refcnt++;
378             }
379         }
380     bad:
381         splx(s);
382         return (error);
383 }
```

route.c

Locate corresponding interface

**340-342**

The function `ifa_ifwithroute` finds the appropriate interface and returning a pointer to its `ifaddr` structure.

## **Allocate memory for routing table entry**

**343-348**

An `rtentry` structure is allocated. Recall that this structure contains pointers to the `ifaddr` structures for the routing tree and the other route. The `rt_flags` are set from the caller's flags, including the `RTF_GATEWAY`.

## **Allocate and copy gateway address**

**349-352**

The `rt_setgate` function ([Figure 19.11](#)) allocates a new `rtentry` structure and its gateway. It then copies `gateway` into the `rt_gateway`, and `rt_gwroute`.

## **Copy destination address**

**353-357**

The destination address (the routing table key) is copied into the `rt_dst` field.

pointed to by rn\_key. If a network mask is supplied, it is ANDed with dst to form the netmask, forming the new key. Otherwise dst is used as the key. This is done logically ANDing dst and netmask is to guarantee that the search key is correctly formed. The search key needs to be ANDed with its mask, so when a search key is created, the search key needs to be ANDed. For example, the interface le0 is assigned an IP address 140.252.12.32 (an alias) to the Ethernet interface le0, with subnet mask 255.255.255.0:

```
bsdi $ ifconfig le0 inet 140.252.12.32 netmask 255.255.255.0
```

The problem is that we've incorrectly specified the key is stored in the routing table we can ve ANDed with the mask:

Destination	Gateway
140.252.12.32	link#1

## Add entry to routing tree

358-366

The rnh\_addaddr function (rn\_addroute from [route.h](#)) takes a destination and mask, to the routing table tree. If EEXIST is returned (i.e., the entry is already present), the entry is updated.

## Store interface pointers

367-369

The ifaddr structure's reference count is incremented when structures are stored.

## **Copy metrics for newly cloned route**

370-371

If the command was RTM\_RESOLVE (not RTM\_SET), copy the cloned entry into the new entry. If the command was RTM\_SET, copy metrics after this function returns.

## **Call interface request function**

372-373

If an ifa\_rtrequest function is defined for this interface, call it to perform additional processing for both the RTM\_SET and RTM\_RESOLVE commands (see section 21.13).

## **Return pointer and increment reference count**

374-378

If the caller wants a copy of the pointer to the new ifaddr structure,

the `rt_refcnt` reference count is incremented from 1 to 2.

## Example: Cloned Routes with Network Masks

The only use of the `rt_genmask` value is with the `cl` command in `rtrequest`. If an `rt_genmask` pointer pointed to by this pointer becomes the network table, [Figure 18.2](#), the cloned routes are for the following example from [[Sklower 1991](#)] provided. This example is in [Exercise 19.2](#).

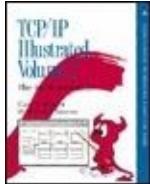
Consider a class B network, say 128.1, that is `0xfffffff00`, the typical value that uses 8 bits for the subnet mask. We need a routing table entry for all possible 254 subnets. Each subnet is directly connected to our host and that knows how to reach it. This is connected.

The easiest solution, assuming the gateway router has a destination of 128.1.0.0 and a mask of `0xfffffff00`, is to create a route for each subnet. The 128.1 network is such that each of the possible 254 subnets has a unique set of characteristics: RTTs, MTUs, delays, and so on. If we look at the routing table for each subnet, we would see that whenever a connection is made to a subnet, there is a corresponding table entry with statistics about that route. It is not practical to create 254 entries by hand using `rtrequest`. A better solution is to use the cloning feature.

One entry is created by the system administrator, and the rest are clones of that entry.

mask of 0xffff0000. Additionally, the RTF\_CLON 0xffffffff00, which differs from the network mask and an entry does not exist for the 128.1.2 sub 0xffff0000 is the best match. A new entry is cre destination of 128.1.2 and a network mask of C host on this subnet is referenced, say 128.1.2.8

---



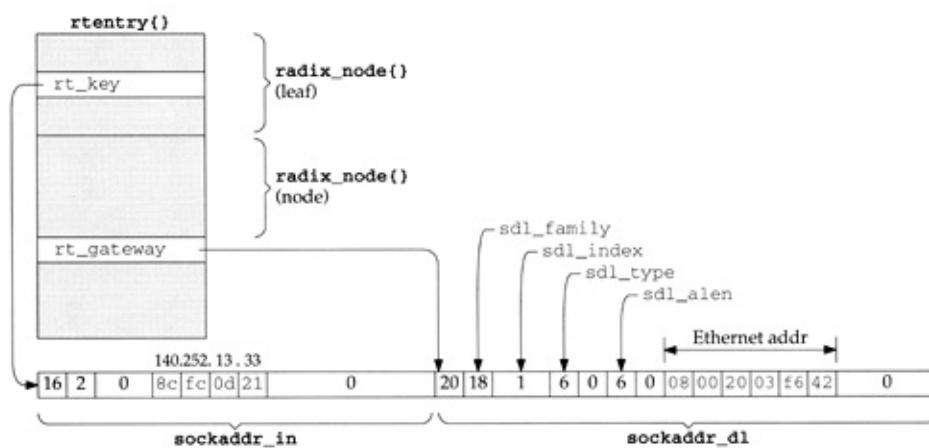
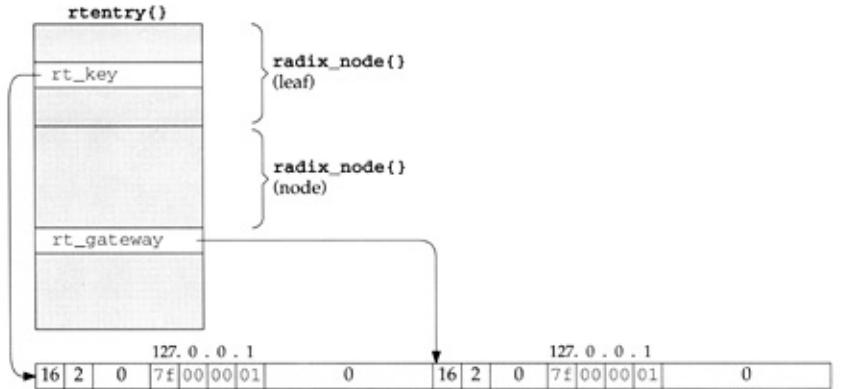
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.5 rt\_setgate Function

Each leaf in the routing tree has a key (`rt_key`, which is just the `rn_key` member of the `radix_node` structure contained at the beginning of the `rtentry` structure), and an associated gateway (`rt_gateway`). Both are socket address structures specified when the routing table entry is created. Memory is allocated for both structures by `rt_setgate`, as shown in Figure 19.10.

**Figure 19.10. Example of routing table keys and associated gateways.**



This example shows two of the entries from [Figure 18.2](#), the ones with keys of 127.0.0.1 and 140.252.13.33. The former's gateway member points to an Internet socket address structure, while the latter's points to a data-link socket address structure that contains an Ethernet address. The former was entered into the routing table by the route system when the system was initialized, and the latter was created by ARP.

We purposely show the two structures pointed to by `rt_key` one right after the other, since they are allocated together by `rt_setgate`, which we show in [Figure 19.11](#).

## Figure 19.11. `rt_setgate` function.

```
384 int  
385 rt_setgate(rt0, dst, gate)  
386 struct rtentry *rt0;  
387 struct sockaddr *dst, *gate;  
388 {  
389     caddr_t new, old;  
390     int      dlen = ROUNDUP(dst->sa_len), glen = ROUNDUP(gate->sa_len);  
391     struct rtentry *rt = rt0;  
392  
393     if (rt->rt_gateway == 0 || glen > ROUNDUP(rt->rt_gateway->sa_len)) {  
394         old = (caddr_t) rt_key(rt);  
395         R_Malloc(new, caddr_t, dlen + glen);  
396         if (new == 0)  
397             return 1;  
398         rt->rt_nodes->rn_key = new;  
399     } else {  
400         new = rt->rt_nodes->rn_key;  
401         old = 0;  
402     }  
403     Bcopy(gate, (rt->rt_gateway = (struct sockaddr *) (new + dlen)), glen);  
404     if (old) {  
405         Bcopy(dst, new, dlen);  
406         Free(old);  
407     }  
408     if (rt->rt_gwroute) {  
409         rt = rt->rt_gwroute;  
410         RTFREE(rt);  
411         rt = rt0;  
412         rt->rt_gwroute = 0;  
413     }  
414     if (rt->rt_flags & RTF_GATEWAY) {  
415         rt->rt_gwroute = rtalloc1(gate, 1);  
416     }  
417     return 0;  
418 }
```

*route.c*

## Set lengths from socket address structures

384-391

dlen is the length of the destination socket address structure, and glen is the length of the gateway socket address structure. The ROUNDUP macro rounds the value up to the next multiple of 4 bytes, but the size of most socket address structures is already a multiple of 4.

## Allocate memory

392-397

If memory has not been allocated for this routing table key and gateway yet, or if glen is greater than the current size of the structure pointed to by rt\_gateway, a new piece of memory is allocated and rn\_key is set to point to the new memory.

## Use memory already allocated for key and gateway

398-401

An adequately sized piece of memory is

already allocated for the key and gateway, so new is set to point to this existing memory.

## **Copy new gateway**

**402**

The new gateway structure is copied and rt\_gateway is set to point to the socket address structure.

## **Copy key from old memory to new memory**

**403-406**

If a new piece of memory was allocated, the routing table key (dst) is copied right before the gateway field that was just copied. The old piece of memory is released.

## **Release gateway routing pointer**

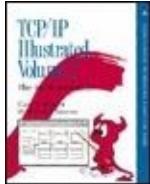
**407-412**

If the routing table entry contains a nonnull `rt_gwroute` pointer, that structure is released by `RTFREE` and the `rt_gwroute` pointer is set to null.

## Locate and store new gateway routing pointer

413-415

If the routing table entry is an indirect route, `rtalloc1` locates the entry for the new gateway, which is stored in `rt_gwroute`. If an invalid gateway is specified for an indirect route, an error is not returned by `rt_setgate`, but the `rt_gwroute` pointer will be null.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.6 rtinit Function

There are four calls to rtinit from the Internet protocols to add or delete routes associated with interfaces.

- in\_control calls rtinit twice when the destination address of a point-to-point interface is set ([Figure 6.21](#)). The first call specifies RTM\_DELETE to delete any existing route to the destination; the second call specifies RTM\_ADD to add the new route.
- in\_ifinit calls rtinit to add a network route for a broadcast network or a host route for a point-to-point link ([Figure](#)

[6.19](#)). If the route is for an Ethernet interface, the RTF\_CLONING flag is automatically set by `in_ifinit`.

- `in_ifscrub` calls `rtinit` to delete an existing route for an interface.

[Figure 19.12](#) shows the first part of the `rtinit` function. The `cmd` argument is always `RTM_ADD` or `RTM_DELETE`.

**Figure 19.12. `rtinit` function: call `rtrequest` to handle command.**

---

```

441 int
442 rtinit(ifa, cmd, flags)
443 struct ifaddr *ifa;
444 int     cmd, flags;
445 {
446     struct rtentry *rt;
447     struct sockaddr *dst;
448     struct sockaddr *deldst;
449     struct mbuf *m = 0;
450     struct rtentry *nrt = 0;
451     int     error;

452     dst = flags & RTF_HOST ? ifa->ifa_dstaddr : ifa->ifa_addr;
453     if (cmd == RTM_DELETE) {
454         if ((flags & RTF_HOST) == 0 && ifa->ifa_netmask) {
455             m = m_get(M_WAIT, MT_SONAME);
456             deldst = mtod(m, struct sockaddr *);
457             rt_maskedcopy(dst, deldst, ifa->ifa_netmask);
458             dst = deldst;
459         }
460         if (rt = rtalloc1(dst, 0)) {
461             rt->rt_refcnt--;
462             if (rt->rt_ifa != ifa) {
463                 if (m)
464                     (void) m_free(m);
465                 return (flags & RTF_HOST ? EHOSTUNREACH
466                               : ENETUNREACH);
467             }
468         }
469     }
470     error = rtrequest(cmd, dst, ifa->ifa_addr, ifa->ifa_netmask,
471                       flags | ifa->ifa_flags, &nrt);
472     if (m)
473         (void) m_free(m);

```

---

route.c

## Get destination address for route

452

If the route is to a host, the destination address is the other end of the point-to-point link. Otherwise we're dealing with a network route and the destination address is the unicast address of the interface (masked with ifa\_netmask).

## Mask network address with network

## mask

453-459

If a route is being deleted, the destination must be looked up in the routing table to locate its routing table entry. If the route being deleted is a network route and the interface has an associated network mask, an mbuf is allocated and the destination address is copied into the mbuf by `rt_maskedcopy`, logically ANDing the caller's address with the mask. `dst` is set to point to the masked copy in the mbuf, and that is the destination looked up in the next step.

## Search for routing table entry

460-469

`rtalloc1` searches the routing table for the destination address. If the entry is found, its reference count is decremented (since `rtalloc1` incremented the reference count). If the pointer to the interface's `ifaddr` in the routing table does not equal the

caller's argument, an error is returned.

## Process request

470-473

rtrequest executes the command, either RTM\_ADD or RTM\_DELETE. When it returns, if an mbuf was allocated earlier, it is released.

[Figure 19.13](#) shows the second half of rtinit.

### Figure 19.13. rtinit function: second half.

```
474     if (cmd == RTM_DELETE && error == 0 && (rt = nrt)) {
475         rt_newaddrmsg(cmd, ifa, error, nrt);
476         if (rt->rt_refcnt <= 0) {
477             rt->rt_refcnt++;
478             rtfree(rt);
479         }
480     }
481     if (cmd == RTM_ADD && error == 0 && (rt = nrt)) {
482         rt->rt_refcnt--;
483         if (rt->rt_ifa != ifa) {
484             printf("rtinit: wrong ifa (%x) was (%x)\n", ifa,
485                   rt->rt_ifa);
486             if (rt->rt_ifa->if_a_rtrequest)
487                 rt->rt_ifa->if_a_rtrequest(RTM_DELETE, rt, SA(0));
488             IFAFREE(rt->rt_ifa);
489             rt->rt_ifa = ifa;
490             rt->rt_ifp = ifa->if_a_ifp;
491             ifa->if_a_refcnt++;
492             if (ifa->if_a_rtrequest)
493                 ifa->if_a_rtrequest(RTM_ADD, rt, SA(0));
494         }
495         rt_newaddrmsg(cmd, ifa, error, nrt);
496     }
497     return (error);
498 }
```

## **Generate routing message on successful delete**

**474-480**

If a route was deleted, and rtrequest returned 0 along with a pointer to the rtentry structure that was deleted (in nrt), a routing socket message is generated by rt\_newaddrmsg. If the reference count is less than or equal to 0, it is incremented and the route is released by rtfree.

## **Successful add**

**481-482**

If a route was added, and rtrequest returned 0 along with a pointer to the rtentry structure that was added (in nrt), the reference count is decremented (since rtrequest incremented it).

## **Incorrect interface**

**483-494**

If the pointer to the interface's ifaddr in the new routing table entry does not equal the caller's argument, an error occurred. Recall that rtrequest determines the ifa pointer that is stored in the new entry by calling `ifa_ifwithroute` ([Figure 19.9](#)). When this error occurs the following steps take place: an error message is output to the console, the `ifa_rtrequest` function is called (if defined) with a command of `RTM_DELETE`, the ifaddr structure is released, the `rt_ifa` pointer is set to the value specified by the caller, the interface reference count is incremented, and the new interface's `ifa_rtrequest` function (if defined) is called with a command of `RTM_ADD`.

## Generate routing message

495

A routing socket message is generated by `rt_newaddrmsg` for the `RTM_ADD` command.

---

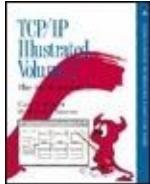
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.7 rtredirect Function

When an ICMP redirect is received, `icmp_input` calls `rtredirect` and then calls `pfctlinput` ([Figure 11.27](#)). This latter function calls `udp_ctlinput` and `tcp_ctlinput`, which go through all the UDP and TCP protocol control blocks. If the PCB is connected to the foreign address that has been redirected, and if the PCB holds a route to that foreign address, the route is released by `rtfree`. The next time any of these control blocks is used to send an IP datagram to that foreign address, `rtalloc` will be called and the destination will be looked up in the routing table, possibly finding a new (redirected) route.

The purpose of rtredirect, the first half of which is shown in [Figure 19.14](#), is to validate the information in the redirect, update the routing table immediately, and then generate a routing socket message.

## Figure 19.14. rtredirect function: validate received redirect.

```
----- route.c
147 int
148 rtredirect(dst, gateway, netmask, flags, src, rtp)
149 struct sockaddr *dst, *gateway, *netmask, *src;
150 int      flags;
151 struct rtentry **rtp;
152 {
153     struct rtentry *rt;
154     int      error = 0;
155     short   *stat = 0;
156     struct rt_addrinfo info;
157     struct ifaddr *ifa;

158     /* verify the gateway is directly reachable */
159     if ((ifa = ifa_ifwithnet(gateway)) == 0) {
160         error = ENETUNREACH;
161         goto out;
162     }
163     rt = rtalloc1(dst, 0);
164     /*
165      * If the redirect isn't from our current router for this dst,
166      * it's either old or wrong.  If it redirects us to ourselves,
167      * we have a routing loop, perhaps as a result of an interface
168      * going down recently.
169      */
170 #define equal(a1, a2) (bcmpl((caddr_t)(a1), (caddr_t)(a2), (a1)->sa_len) == 0)
171     if (!(flags & RTF_DONE) && rt &&
172         (!equal(src, rt->rt_gateway) || rt->rt_ifa != ifa))
173         error = EINVAL;
174     else if (ifa_ifwithaddr(gateway))
175         error = EHOSTUNREACH;
176     if (error)
177         goto done;
178     /*
179      * Create a new entry if we just got back a wildcard entry
180      * or if the lookup failed.  This is necessary for hosts
181      * which use routing redirects generated by smart gateways
182      * to dynamically build the routing tables.
183      */
184     if ((rt == 0) || (rt->rt_mask && rt->rt_mask->sa_len < 2))
185         goto create;
```

---

```
----- route.c
```

147-157

The arguments are dst, the destination IP address of the datagram that caused the redirect (HD in [Figure 8.18](#)); gateway, the IP address of the router to use as the new gateway field for the destination (R2 in [Figure 8.18](#)); netmask, which is a null pointer; flags, which is RTF\_GATEWAY and RTF\_HOST; src, the IP address of the router that sent the redirect (R1 in [Figure 8.18](#)); and rtp, which is a null pointer. We indicate that netmask and rtp are both null pointers when called by icmp\_input, but these arguments might be nonnull when called from other protocols.

## New gateway must be directly connected

158-162

The new gateway must be directly connected or the redirect is invalid.

## Locate routing table entry for destination and validate redirect

rtalloc1 searches the routing table for a route to the destination. The following conditions must all be true, or the redirect is invalid and an error is returned. Notice that icmp\_input ignores any error return from rtredirect. ICMP does not generate an error in response to an invalid redirectit just ignores it.

- the RTF\_DONE flag must not be set;
- rtalloc must have located a routing table entry for dst;
- the address of the router that sent the redirect (src) must equal the current rt\_gateway for the destination;
- the interface for the new gateway (the ifa returned by ifa\_ifwithnet) must equal the current interface for the destination (rt\_ifa), that is, the new gateway must be on the same network as the current gateway; and
- the new gateway cannot redirect this host to itself, that is, there cannot exist

an attached interface with a unicast address or a broadcast address equal to gateway.

## Must create a new route

178-185

If a route to the destination was not found, or if the routing table entry that was located is the default route, a new entry is created for the destination. As the comment indicates, a host with access to multiple routers can use this feature to learn of the correct router when the default is not correct. The test for finding the default route is whether the routing table entry has an associated mask and if the length field of the mask is less than 2, since the mask for the default route is rn\_zeros ([Figure 18.35](#)).

[Figure 19.15](#) shows the second half of this function.

**Figure 19.15. rtredirect function: second**

# half.

```
186  /*
187   * Don't listen to the redirect if it's
188   * for a route to an interface.
189   */
190  if (rt->rt_flags & RTF_GATEWAY) {
191      if (((rt->rt_flags & RTF_HOST) == 0) && (flags & RTF_HOST)) {
192          /*
193           * Changing from route to net => route to host.
194           * Create new route, rather than smashing route to net.
195           */
196          create:
197              flags |= RTF_GATEWAY | RTF_DYNAMIC;
198              error = rtrequest((int) RTM_ADD, dst, gateway,
199                                netmask, flags,
200                                (struct rtentry **) 0);
201              stat = &rtstat.rts_dynamic;
202      } else {
203          /*
204           * Smash the current notion of the gateway to
205           * this destination. Should check about netmask!!!
206           */
207          rt->rt_flags |= RTF_MODIFIED;
208          flags |= RTF_MODIFIED;
209          stat = &rtstat.rts_newgateway;
210          rt_setgate(rt, rt_key(rt), gateway);
211      }
212  } else
213      error = EHOSTUNREACH;
214 done:
215  if (rt) {
216      if (rtp && !error)
217          *rtp = rt;
218      else
219          rtfree(rt);
220  }
221 out:
222  if (error)
223      rtstat.rts_badredirect++;
224  else if (stat != NULL)
225      (*stat)++;
226  bzero((caddr_t) & info, sizeof(info));
227  info.rti_info[RTAX_DST] = dst;
228  info.rti_info[RTAX_GATEWAY] = gateway;
229  info.rti_info[RTAX_NETMASK] = netmask;
230  info.rti_info[RTAX_AUTHOR] = src;
231  rt_missmsg(RTM_REDIRECT, &info, flags, error);
232 }
```

route.c

## Create new host route

186-195

If the current route to the destination is a

network route and the redirect is a host redirect and not a network redirect, a new host route is created for the destination and the existing network route is left alone. We mentioned that the flags argument always specifies RTF\_HOST since the Net/3 ICMP considers all received redirects as host redirects.

## Create route

196-201

rtrequest creates the new route, setting the RTF\_GATEWAY and RTF\_DYNAMIC flags. The netmask argument is a null pointer, since the new route is a host route with an implied mask of all one bits. stat points to a counter that is incremented later.

## Modify existing host route

202-211

This code is executed when the current route to the destination is already a host

route. A new entry is not created, but the existing entry is modified. The RTF\_MODIFIED flag is set and `rt_setgate` changes the `rt_gateway` field of the routing table entry to the new gateway address.

## **Ignore if destination is directly connected**

212-213

If the current route to the destination is a direct route (the RTF\_GATEWAY flag is not set), it is a redirect for a destination that is already directly connected.  
EHOSTUNREACH is returned.

## **Return pointer and increment statistic**

214-225

If a routing table entry was located, it is either returned (if `rtp` is nonnull and there were no errors) or released by `rtfree`. The appropriate statistic is incremented.

## Generate routing message

226-232

An `rt_addrinfo` structure is cleared and a routing socket message is generated by `rt_missmsg`. This message is sent by `raw_input` to any processes interested in the redirect.

---

Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)

[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Router Messages

### 19.8 Routing Message Structures

Routing messages consist of a fixed-length header followed by up to eight socket address structures. The fixed-length header is one of the following structures:

- rt\_msghdr
- if\_msghdr
- ifa\_msghdr

[Figure 18.11](#) provided an overview of which functions generated the different messages and [Figure 18.12](#) showed which structure is used by each message type. The first three members of the three structures have the same data type and meaning: the message identifier,

length, version, and type. This allows the receiver of the message to decode the message. Also, each structure has a member that encodes which of the eight potential socket address structures follow the structure (a bitmask): the rtm\_addrs, ifm\_addrs, and ifam\_addrs members.

Figure 19.16 shows the most common of the structures, rt\_msghdr. The RTM\_IFINFO message uses an if\_msghdr structure, shown in Figure 19.17. The RTM\_NEWADDR and RTM\_DELADDR messages use an ifa\_msghdr structure, shown in Figure 19.18.

## Figure 19.16. rt\_msghdr structure.

```
139 struct rt_msghdr {  
140     u_short rtm_msghlen;          /* to skip over non-understood messages */  
141     u_char rtm_version;          /* future binary compatibility */  
142     u_char rtm_type;             /* message type */  
143     u_short rtm_index;           /* index for associated ifp */  
144     int rtm_flags;               /* flags, incl. kern & message, e.g. DONE */  
145     int rtm_addrs;               /* bitmask identifying sockaddr in msg */  
146     pid_t rtm_pid;               /* identify sender */  
147     int rtm_seq;                 /* for sender to identify action */  
148     int rtm_errno;               /* why failed */  
149     int rtm_use;                 /* from rtentry */  
150     u_long rtm_inits;            /* which metrics we are initializing */  
151     struct rt_metrics rtm_rmx;   /* metrics themselves */  
152 };
```

## Figure 19.17. if\_msghdr structure.

```

235 struct if_msghdr {
236     u_short ifm_msrlen;           /* to skip over non-understood messages */
237     u_char ifm_version;          /* future binary compatibility */
238     u_char ifm_type;             /* message type */
239     int     ifm_addrs;            /* like rtm_addrs */
240     int     ifm_flags;            /* value of if_flags */
241     u_short ifm_index;           /* index for associated ifp */
242     struct if_data ifm_data;     /* statistics and other data about if */
243 };

```

-if.h

## Figure 19.18. ifa\_msghdr structure.

```

248 struct ifa_msghdr {
249     u_short ifam_msrlen;          /* to skip over non-understood messages */
250     u_char ifam_version;          /* future binary compatibility */
251     u_char ifam_type;             /* message type */
252     int     ifam_addrs;            /* like rtm_addrs */
253     int     ifam_flags;            /* value of ifa_flags */
254     u_short ifam_index;           /* index for associated ifp */
255     int     ifam_metric;           /* value of ifa_metric */
256 };

```

-if.h

Note that the first three members across the three different structures have the same data types and meanings.

The three variables rtm\_addrs, ifm\_addrs, and ifam\_addrs are bitmasks defining which socket address structures follow the header. [Figure 19](#) shows the constants used with these bitmasks.

## Figure 19.19. Constants used to refer to members of rti\_info array.

Bitmask		Array index		Name in rtsock.c	Description
Constant	Value	Constant	Value		
<code>RTA_DST</code>	0x01	<code>RTAX_DST</code>	0	<code>dst</code>	destination socket address structure
<code>RTA_GATEWAY</code>	0x02	<code>RTAX_GATEWAY</code>	1	<code>gate</code>	gateway socket address structure
<code>RTA_NETMASK</code>	0x04	<code>RTAX_NETMASK</code>	2	<code>netmask</code>	netmask socket address structure
<code>RTA_GENMASK</code>	0x08	<code>RTAX_GENMASK</code>	3	<code>genmask</code>	cloning mask socket address structure
<code>RTA_IFP</code>	0x10	<code>RTAX_IFP</code>	4	<code>ifpaddr</code>	interface name socket address structure
<code>RTA_IFA</code>	0x20	<code>RTAX_IFA</code>	5	<code>ifaaddr</code>	interface address socket address structure
<code>RTA_AUTHOR</code>	0x40	<code>RTAX_AUTHOR</code>	6		socket address structure for author of redirect
<code>RTA_BRD</code>	0x80	<code>RTAX_BRD</code>	7	<code>brdaddr</code>	broadcast or point-to-point destination address
		<code>RTAX_MAX</code>	8		#elements in an <code>rti_info[]</code> array

The bitmask value is always the constant 1 left shifted by the number of bits specified by the array index. For example, 0x20 (RTA\_IFA) is 1 left shifted by five bits (RTAX\_IFA). We'll see this fact used in code.

The socket address structures that are present occur in order of increasing array index, one right after the other. For example, if the bitmask is 0x0f the first socket address structure contains the destination, followed by the gateway, followed by the network mask, followed by the broadcast address.

The array indexes in [Figure 19.19](#) are used within the kernel to refer to its `rt_addrinfo` structure, shown in [Figure 19.20](#). This structure holds the same bitmasks that we described, indicating which addresses are present, and pointers to those socket address structures.

## Figure 19.20. `rt_addrinfo` structure: encode \

## addresses are present and pointers to the

```
199 struct rt_addrinfo {  
200     int      rti_addrs;           /* bitmask, same as rtm_addrs */  
201     struct sockaddr *rti_info[RTAX_MAX];  
202 };
```

For example, if the RTA\_GATEWAY bit is set in the rti\_addrs member, then the member rti\_info[RTAX\_GATEWAY] is a pointer to a socket address structure containing the gateway's address. In the case of the Internet protocols, the socket address structure is a sockaddr\_in containing the gateway's IP address.

The fifth column in [Figure 19.19](#) shows the names used for the corresponding members of an rti\_info array throughout the file rtsock.c. These definitely look like

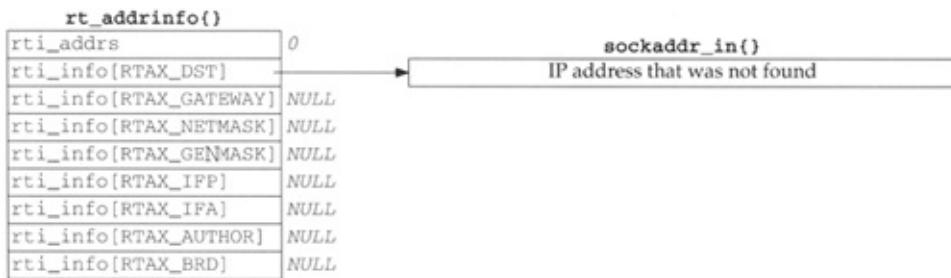
```
#define dst    info.rti_info[RTAX_GATEWAY]
```

We'll encounter these names in many of the source files later in this chapter. The RTAX\_AUTHOR element is not assigned a name because it is never passed from a process to the kernel.

We've already encountered this rt\_addrinfo structure twice: in rtalloc1 ([Figure 19.2](#)) and rtredirect ([Figure 19.3](#)).

19.14). Figure 19.21 shows the format of this structure when built by rtalloc1, after a routing lookup fails, when rt\_missmsg is called.

**Figure 19.21. rt\_addrinfo structure passed rtalloc1 to rt\_missmsg.**

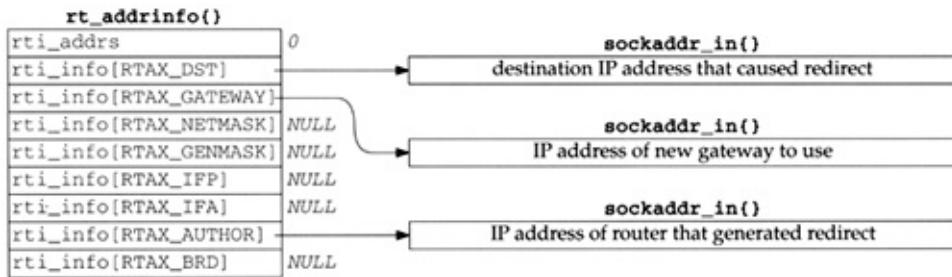


All the unused pointers are null because the str is set to 0 before it is used. Also note that the `rti_addrs` member is not initialized with the appropriate bitmask because when this structure is used within the kernel, a null pointer in the `rti_info` array indicates a nonexistent socket address structure. The bitmask is needed only for message exchange between a process and the kernel.

Figure 19.22 shows the format of the structure passed by rtredirect when it calls rt\_missmsg.

**Figure 19.22. rt\_addrinfo structure passed by rtredirect when it calls rt\_missmsg.**

## rtredirect to rt\_missmsg.



The following sections show how these structures are placed into the messages sent to a process.

[Figure 19.23](#) shows the `route_cb` structure, which we'll encounter in the following sections. It contains four counters; one each for the IP, XNS, and ISO protocols, and an "any" counter. Each counter is the number of routing sockets currently in existence in that domain.

**Figure 19.23. route\_cb structure: counters for routing socket listeners.**

```
203 struct route_cb {
204     int     ip_count;          /* IP */
205     int     ns_count;          /* XNS */
206     int     iso_count;          /* ISO */
207     int     any_count;          /* sum of above three counters */
208 };
```

By keeping track of the number of routing socket listeners, the kernel avoids building a routing message and calling raw\_input to send the message when there aren't any processes waiting for a message.

---

## Chapter 19. Routing Requests and Responses

### 19.9 rt\_missmsg Function

The function `rt_missmsg`, shown in [Figure 19.2](#), takes a pointer to the structures shown in [Figures 19.21](#) and [19.22](#), creates a new mbuf chain, builds a corresponding variable-length message mbuf chain, and then calls `raw_input` to pass the message to the appropriate routing sockets.

**Figure 19.24. `rt_missmsg` function**

```

516 void
517 rt_missmsg(type, rtinfo, flags, error)
518 int     type, flags, error;
519 struct rt_addrinfo *rtinfo;
520 {
521     struct rt_msghdr *rtm;
522     struct mbuf *m;
523     struct sockaddr *sa = rtinfo->rti_info[RTAX_DST];
524     if (route_cb.any_count == 0)
525         return;
526     m = rt_msg1(type, rtinfo);
527     if (m == 0)
528         return;
529     rtm = mtod(m, struct rt_msghdr *);
530     rtm->rtm_flags = RTF_DONE | flags;
531     rtm->rtm_errno = error;
532     rtm->rtm_addrs = rtinfo->rti_addrs;
533     route_proto.sp_protocol = sa ? sa->sa_family : 0;
534     raw_input(m, &route_proto, &route_src, &route_dst);
535 }

```

rtsock.c

## 516-525

If there aren't any routing socket listeners, the immediately.

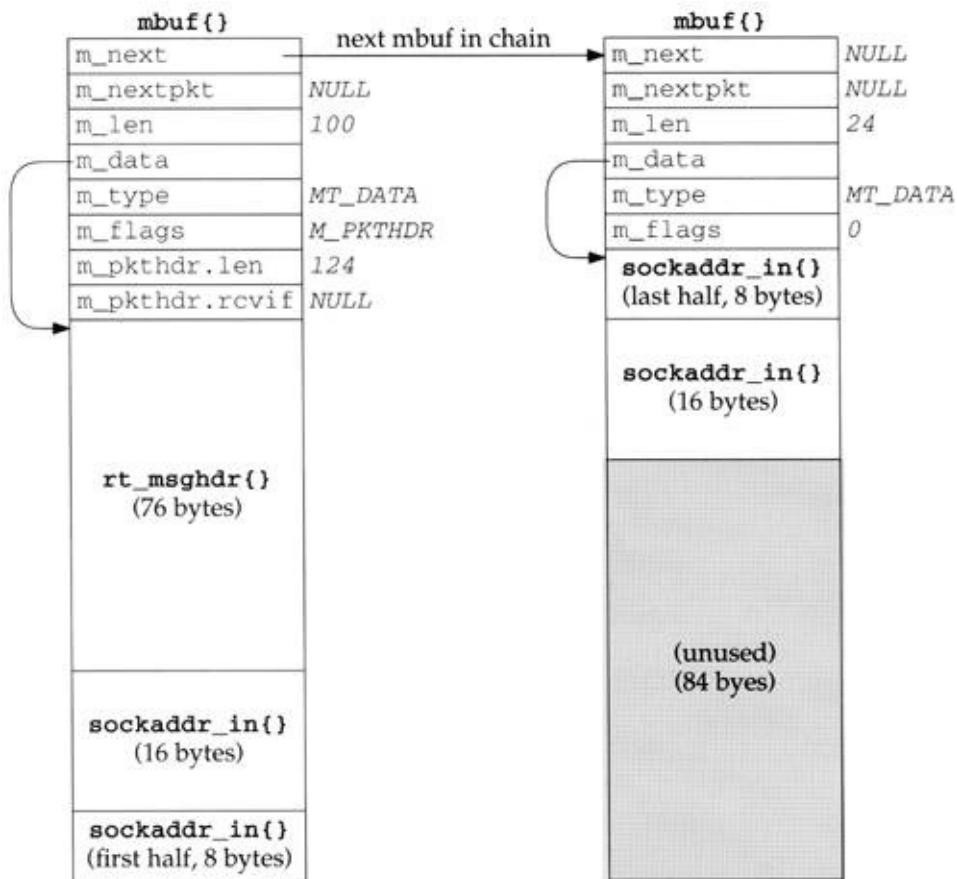
## Build message in mbuf chain

## 526-528

`rt_msg1` ([Section 19.12](#)) builds the appropriate mbuf chain, and returns the pointer to the chain. It shows an example of the resulting mbuf chain, the `rt_addrinfo` structure from [Figure 19.22](#). The information in the `rti_info` field must be in an mbuf chain because `raw_input` calls `mbuf_chain_append` to append the mbuf chain to a socket's receive buffer.

**Figure 19.25. Mbuf chain built by `rt_msg1`**

**Figure 19.22.**



## Finish building message

529-532

The two members rtm\_flags and rtm\_errno are passed by the caller. The rtm\_addrs member is rti\_addrs value. We showed this value as 0 in [F 19.22](#), but rt\_msg1 calculates and stores the a based on which pointers in the rti\_info array ar

## Set protocol of message, call raw\_input

533-534

The final three arguments to raw\_input specify source, and destination of the routing message structures are initialized as

```
struct sockaddr route_dst = {  
    struct sockaddr route_src = {  
        struct sockproto route_proto =
```

The first two structures are never modified by the sockproto structure, shown in [Figure 19.26](#), is constant before.

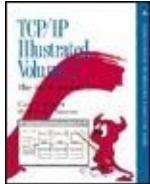
**Figure 19.26. sockproto structure**

```
128 struct sockproto {  
129     u_short sp_family;           /* address family */  
130     u_short sp_protocol;         /* protocol */  
131 };
```

The family is never changed from its initial value of AF\_ROUTE. The protocol is set each time raw\_input is called. The caller creates a routing socket by calling socket, the argument sp\_protocol (which is a pointer to a protocol) specifies the protocol in which the message is to be transmitted. The caller of raw\_input sets the sp\_protocol member to the value of the protocol argument.

route\_proto structure to the protocol of the route. In the case of rt\_missmsg, it is set to the sa\_family member of the socket address structure (if specified by the caller). Figures 19.21 and 19.22 would be AF\_INET.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.10 rt\_ifmsg Function

In [Figure 4.30](#) we saw that `if_up` and `if_down` both call `rt_ifmsg`, shown in [Figure 19.27](#), to generate a routing socket message when an interface goes up or down.

**Figure 19.27. `rt_ifmsg` function.**

```
540 void  
541 rt_ifmsg(ifp)  
542 struct ifnet *ifp;  
543 {  
544     struct if_msghdr *ifm;  
545     struct mbuf *m;  
546     struct rt_addrinfo info;  
547     if (route_cb.any_count == 0)  
548         return;  
549     bzero((caddr_t) &info, sizeof(info));  
550     m = rt_msgl(RTM_IFINFO, &info);  
551     if (m == 0)  
552         return;  
553     ifm = mtod(m, struct if_msghdr *);  
554     ifm->ifm_index = ifp->if_index;  
555     ifm->ifm_flags = ifp->if_flags;  
556     ifm->ifm_data = ifp->if_data; /* structure assignment */  
557     ifm->ifm_addrs = 0;  
558     route_proto.sp_protocol = 0;  
559     raw_input(m, &route_proto, &route_src, &route_dst);  
560 }
```

rtsock.c

## 547-548

If there aren't any routing socket listeners, the function returns immediately.

## Build message in mbuf chain

## 549-552

An `rt_addrinfo` structure is set to 0 and `rt_msgl` builds an appropriate message in an mbuf chain. Notice that all socket address pointers in the `rt_addrinfo` structure are null, so only the fixed-length `if_msghdr` structure becomes the routing message; there are no addresses.

## Finish building message

553-557

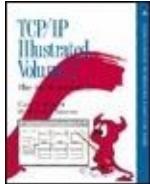
The interface's index, flags, and if\_data structure are copied into the message in the mbuf and the ifm\_addrs bitmask is set to 0.

## Set protocol of message, call raw\_input

558-559

The protocol of the routing message is set to 0 because this message can apply to all protocol suites. It is a message about an interface, not about some specific destination. raw\_input delivers the message to the appropriate listeners.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.11 rt\_newaddrmsg Function

In [Figure 19.13](#) we saw that rtinit calls `rt_newaddrmsg` with a command of `RTM_ADD` or `RTM_DELETE` when an interface has an address added or deleted. [Figure 19.28](#) shows the first half of the function.

**Figure 19.28. `rt_newaddrmsg` function:  
first half: create `ifa_msghdr` message.**

```
569 void
570 rt_newaddrmsg(cmd, ifa, error, rt)
571 int cmd, error;
572 struct ifaddr *ifa;
573 struct rtentry *rt;
574 {
575     struct rt_addrinfo info;
576     struct sockaddr *sa;
577     int pass;
578     struct mbuf *m;
579     struct ifnet *ifp = ifa->ifa_ifp;
580     if (route_cb.any_count == 0)
581         return;
582     for (pass = 1; pass < 3; pass++) {
583         bzero((caddr_t) & info, sizeof(info));
584         if ((cmd == RTM_ADD && pass == 1) ||
585             (cmd == RTM_DELETE && pass == 2)) {
586             struct ifa_msghdr *ifam;
587             int ncmd = cmd == RTM_ADD ? RTM_NEWSADDR : RTM_DELADDR;
588             ifaaddr = sa = ifa->ifa_addr;
589             ifpaddr = ifp->if_addrlist->ifa_addr;
590             netmask = ifa->ifa_netmask;
591             brdaddr = ifa->ifa_dstaddr;
592             if ((m = rt_msgl(ncmd, &info)) == NULL)
593                 continue;
594             ifam = mtod(m, struct ifa_msghdr *);
595             ifam->ifam_index = ifp->if_index;
596             ifam->ifam_metric = ifa->ifa_metric;
597             ifam->ifam_flags = ifa->ifa_flags;
598             ifam->ifam_addrs = info.rti_addrs;
599         }

```

rtsock.c

## 580-581

If there aren't any routing socket listeners, the function returns immediately.

## Generate two routing messages

## 582

The for loop iterates twice because two messages are generated. If the command is RTM\_ADD, the first message is of type RTM\_NEWSADDR and the second message

is of type RTM\_ADD. If the command is RTM\_DELETE, the first message is of type RTM\_DELETE and the second message is of type RTM\_DELADDR. The RTM\_NEWADDR and RTM\_DELADDR messages are built from an ifa\_msghdr structure, while the RTM\_ADD and RTM\_DELETE messages are built from an rt\_msghdr structure. The function generates two messages because one message provides information about the interface and the other about the addresses.

583

An rt\_addrinfo structure is set to 0.

## Generate message with up to four addresses

588-591

Pointers to four socket address structures containing information about the interface address that has been added or deleted are stored in the rti\_info array. Recall from

[Figure 19.19](#) that ifaaddr, ifpaddr, netmask, and brdaddr reference elements in the rti\_info array in info. rt\_msg1 builds the appropriate message in an mbuf chain. Notice that sa is set to point to the ifa\_addr structure, and we'll see at the end of the function that the family of this socket address structure becomes the protocol of the routing message.

Remaining members of the ifa\_msghdr structure are filled in with the interface's index, metric, and flags, along with the bitmask set by rt\_msg1.

[Figure 19.29](#) shows the second half of rt\_newaddrmsg, which creates an rt\_msghdr message with information about the routing table entry that was added or deleted.

**Figure 19.29. rt\_newaddrmsg function: second half, create rt\_msghdr message.**

```

600      if ((cmd == RTM_ADD && pass == 2) ||
601          (cmd == RTM_DELETE && pass == 1)) {
602          struct rt_msghdr *rtm;
603
604          if (rt == 0)
605              continue;
606          netmask = rt_mask(rt);
607          dst = sa = rt_key(rt);
608          gate = rt->rt_gateway;
609          if ((m = rt_msgh1(cmd, &info)) == NULL)
610              continue;
611          rtm = mtod(m, struct rt_msghdr *);
612          rtm->rtm_index = ifp->if_index;
613          rtm->rtm_flags |= rt->rt_flags;
614          rtm->rtm_errno = error;
615          rtm->rtm_addrs = info.rti_addrs;
616          route_proto.sp_protocol = sa ? sa->sa_family : 0;
617          raw_input(m, &route_proto, &route_src, &route_dst);
618      }
619  }

```

— *rtsock.c*

## Build message

*600-609*

Pointers to three socket address structures are stored in the `rti_info` array: the `rt_mask`, `rt_key`, and `rt_gateway` structures. `sa` is set to point to the destination address, and its family becomes the protocol of the routing message. `rt_msg1` builds the appropriate message in an mbuf chain.

Additional fields in the `rt_msghdr` structure are filled in, including the bitmask set by `rt_msg1`.

## **Set protocol of message, call raw\_input**

616-619

The protocol of the routing message is set and raw\_input passes the message to the appropriate listeners. The function returns after two iterations through the loop.

---

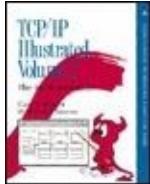
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.12 rt\_msg1 Function

The functions described in the previous three sections each called `rt_msg1` to build the appropriate routing message. In [Figure 19.25](#) we showed the mbuf chain that was built by `rt_msg1` from the `rt_msghdr` and `rt_addrinfo` structures in [Figure 19.22](#). [Figure 19.30](#) shows the function.

**Figure 19.30. `rt_msg1` function: obtain and initialize mbuf.**

---

```

399 static struct mbuf *
400 rt_msgl(type, rtinfo)
401 int      type;
402 struct rt_addrinfo *rtinfo;
403 {
404     struct rt_msghdr *rtm;
405     struct mbuf *m;
406     int      i;
407     struct sockaddr *sa;
408     int      len, dlen;
409
410     m = m_gethdr(M_DONTWAIT, MT_DATA);
411     if (m == 0)
412         return (m);
413     switch (type) {
414
415         case RTM_DELADDR:
416         case RTM_NEWADDR:
417             len = sizeof(struct ifa_msghdr);
418             break;
419
420         case RTM_IFINFO:
421             len = sizeof(struct if_msghdr);
422             break;
423
424         default:
425             len = sizeof(struct rt_msghdr);
426         }
427     if (len > MHLEN)
428         panic("rt_msgl");
429     m->m_pkthdr.len = m->m_len = len;
430     m->m_pkthdr.rcvif = 0;
431     rtm = mtod(m, struct rt_msghdr *);
432     bzero((caddr_t) rtm, len);
433
434     for (i = 0; i < RTAX_MAX; i++) {
435         if ((sa = rtinfo->rti_info[i]) == NULL)
436             continue;
437         rtinfo->rti_addrs |= (1 << i);
438         dlen = ROUNDUP(sa->sa_len);
439         m_copyback(m, len, dlen, (caddr_t) sa);
440         len += dlen;
441     }
442     if (m->m_pkthdr.len != len)
443         m_freem(m);
444     return (NULL);
445 }

```

---

rtsock.c

## Get mbuf and determine fixed size of message

399-422

An mbuf with a packet header is obtained

and the length of the fixed-size message is stored in len. Two of the message types in [Figure 18.9](#) use an ifa\_msghdr structure, one uses an if\_msghdr structure, and the remaining nine use an rt\_msghdr structure.

## Verify structure fits in mbuf

423-424

The size of the fixed-length structure must fit entirely within the data portion of the packet header mbuf, because the mbuf pointer is cast to a structure pointer using mtod and the structure is then referenced through the pointer. The largest of the three structures is if\_msghdr, which at 84 bytes is less than MHLEN (100).

## Initialize mbuf packet header and zero structure

425-428

The two fields in the packet header are initialized and the structure in the mbuf is

set to 0.

## Copy socket address structures into mbuf chain

429-436

The caller passes a pointer to an `rt_addrinfo` structure. The socket address structures corresponding to all the nonnull pointers in the `rti_info` are copied into the mbuf by `m_copyback`. The value 1 is left shifted by the `RTAX_xxx` index to generate the corresponding `RTA_xxx` bitmask ([Figure 19.19](#)), and each individual bitmask is logically ORed into the `rti_addrs` member, which the caller can store on return into the corresponding member of the message structure. The `ROUNDUP` macro rounds the size of each socket address structure up to the next multiple of 4 bytes.

437-440

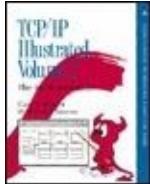
If, when the loop terminates, the length in the mbuf packet header does not equal

`len`, the function `m_copyback` wasn't able to obtain a required mbuf.

## Store length, version, and type

`441-445`

The length, version, and message type are stored in the first three members of the message structure. Again, all three `xxx_msghdr` structures start with the same three members, so this code works with all three structures even though the pointer `rtm` is a pointer to an `rt_msghdr` structure.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.13 rt\_msg2 Function

rt\_msg1 constructs a routing message in an mbuf chain, and the three functions that called it then called raw\_input to append the mbuf chain to one or more socket's receive buffer. rt\_msg2 is differentit builds a routing message in a memory buffer, not an mbuf chain, and has as an argument a pointer to a walkarg structure that is used when rt\_msg2 is called by the two functions that handle the sysctl system call for the routing domain. rt\_msg2 is called in two different scenarios:

#### 1. from route\_output to process the

## **RTM\_GET command, and**

- from sysctl\_dumpentry and sysctl\_iflist to process a sysctl system call.

Before looking at rt\_msg2, [Figure 19.31](#) shows the walkarg structure that is used in scenario 2. We go through all these members as we encounter them.

**Figure 19.31. walkarg structure: used with the sysctl system call in the routing domain.**

```
41 struct walkarg {                                     —rtsock.c
42     int      w_op;          /* NET_RT_XXX */
43     int      w_arg;          /* RTF_XXX for FLAGS, if_index for IFLIST */
44     int      w_given;        /* size of process' buffer */
45     int      w_needed;       /* #bytes actually needed (at end) */
46     int      w_tmemsize;     /* size of buffer pointed to by w_tmem */
47     caddr_t  w_where;        /* ptr to process' buffer (maybe null) */
48     caddr_t  w_tmem;         /* ptr to our malloc'ed buffer */
49 };
                                     —rtsock.c
```

[Figure 19.32](#) shows the first half of the rt\_msg2 function. This portion is similar to the first half of rt\_msg1.

**Figure 19.32. rt\_msg2 function: copy socket address structures.**

```
446 static int
447 rt_msg2(type, rtinfo, cp, w)
448 int      type;
449 struct rt_addrinfo *rtinfo;
450 caddr_t cp;
451 struct walkarg *w;
452 {
453     int      i;
454     int      len, dlen, second_time = 0;
455     caddr_t cp0;
456     rtinfo->rti_addrs = 0;
457 again:
458     switch (type) {
459     case RTM_DELADDR:
460     case RTM_NEWADDR:
461         len = sizeof(struct ifa_msghdr);
462         break;
463     case RTM_IFINFO:
464         len = sizeof(struct if_msghdr);
465         break;
466     default:
467         len = sizeof(struct rt_msghdr);
468     }
469     if (cp0 == cp)
470         cp += len;
471     for (i = 0; i < RTAX_MAX; i++) {
472         struct sockaddr *sa;
473         if ((sa = rtinfo->rti_info[i]) == 0)
474             continue;
475         rtinfo->rti_addrs |= (1 << i);
476         dlen = ROUNDUP(sa->sa_len);
477         if (cp) {
478             bcopy((caddr_t) sa, cp, (unsigned) dlen);
479             cp += dlen;
480         }
481         len += dlen;
482     }
```

## 446-455

Since this function stores the resulting message in a memory buffer, the caller specifies the start of that buffer in the cp argument. It is the caller's responsibility to ensure that the buffer is large enough for the message that is generated. To help the caller determine this size, if the cp argument is null, rt\_msg2 doesn't store anything but processes the input and

returns the total number of bytes required to hold the result. We'll see that `route_output` uses this feature and calls this function twice: first to determine the size and then to store the result, after allocating a buffer of the correct size. When `rt_msg2` is called by `route_output`, the final argument is null. This final argument is nonnull when called as part of the `sysctl` system call processing.

## Determine size of structure

458-470

The size of the fixed-length message structure is set based on the message type. If the `cp` pointer is nonnull, it is incremented by this size.

## Copy socket address structures

471-482

The for loop goes through the `rti_info` array, and for each element that is a nonnull pointer it sets the appropriate bit

in the rti\_addrs bitmask, copies the socket address structure (if cp is nonnull), and updates the length.

Figure 19.33 shows the second half of rt\_msg2, most of which handles the optional walkarg structure.

### Figure 19.33. rt\_msg2 function: handle optional walkarg argument.

```
-----rtsock.c
483     if (cp == 0 && w != NULL && !second_time) {
484         struct walkarg *rw = w;
485
486         rw->w_needed += len;
487         if (rw->w_needed <= 0 && rw->w_where) {
488             if (rw->w_tmemsize < len) {
489                 if (rw->w_tmem)
490                     free(rw->w_tmem, M_RTABLE);
491                 if (rw->w_tmem = (caddr_t)
492                     malloc(len, M_RTABLE, M_NOWAIT))
493                     rw->w_tmemsize = len;
494             }
495             if (rw->w_tmem) {
496                 cp = rw->w_tmem;
497                 second_time = 1;
498                 goto again;
499             } else
500                 rw->w_where = 0;
501         }
502         if (cp) {
503             struct rt_msghdr *rtm = (struct rt_msghdr *) cp0;
504             rtm->rtm_version = RTM_VERSION;
505             rtm->rtm_type = type;
506             rtm->rtm_msflen = len;
507         }
508     return (len);
509 }
```

-----rtsock.c

483-484

This if statement is true only when a

pointer to a walkarg structure was passed and this is the first loop through the function. The variable second\_time was initialized to 0 but can be set to 1 within this if statement, and a jump made back to the label again in [Figure 19.32](#). The test for cp being a null pointer is superfluous since whenever the w pointer is nonnull, the cp pointer is null, and vice versa.

## Check if data to be stored

485-486

w\_needed is incremented by the size of the message. This variable is initialized to 0 minus the size of the user's buffer to the sysctl function. For example, if the buffer size is 500 bytes, w\_needed is initialized to 500. As long as it remains negative, there is room in the buffer. w\_where is a pointer to the buffer in the calling process. It is null if the process doesn't want the resultthe process just wants sysctl to return the size of the result, so the process can allocate a buffer and call sysctl again. rt\_msg2 doesn't copy the data back to the

process that is up to the caller but if the `w_where` pointer is null, there's no need for `rt_msg2` to malloc a buffer to hold the result and loop back through the function again, storing the result in this buffer. There are really five different scenarios that this function handles, summarized in [Figure 19.34](#).

**Figure 19.34. Summary of different scenarios for `rt_msg2`.**

called from	cp	w	w.w_where	second_time	Description
route_output	null	null			wants return length
	nonnull	null			wants result
sysctl_rtable	null	nonnull	null	0	process wants return length
	null	nonnull	nonnull	0	first time around to calculate length
	nonnull	nonnull	nonnull	1	second time around to store result

**Allocate buffer first time or if message length increases**

487-493

`w_tmemszie` is the size of the buffer pointed to by `w_tmem`. It is initialized to 0 by `sysctl_rtable`, so the first time `rt_msg2` is called for a given `sysctl` request, the buffer must be allocated. Also, if the size

of the result increases, the existing buffer must be released and a new (larger) buffer allocated.

## **Go around again and store result**

**494-499**

If `w_tmem` is nonnull, a buffer already exists or one was just allocated. `cp` is set to point to this buffer, `second_time` is set to 1, and a jump is made to `again`. The if statement at the beginning of this figure won't be true during this second pass, since `second_time` is now 1. If `w_tmem` is null, the call to `malloc` failed, so the pointer to the buffer in the process is set to null, preventing anything from being returned.

## **Store length, version, and type**

**502-509**

If `cp` is nonnull, the first three elements of the message header are stored. The function returns the length of the

message.

---

**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)

## Chapter 19. Routing Requests and Rou

### 19.14 sysctl\_rtable Function

This function handles the sysctl system call on socket. It is called by net\_sysctl as shown in Figure 19.35.

Before going through the source code, Figure 19.35 shows a typical use of this system call with respect to the routing table. This example is from the arp program.

**Figure 19.35. Example of sysctl with routing table**

---

```
int      mib[6];
size_t   needed;
char     *buf, *lim, *next;
struct rt_msghdr *rtm;

mib[0] = CTL_NET;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = AF_INET;      /* address family; can be 0 */
mib[4] = NET_RT_FLAGS; /* operation */
mib[5] = RTP_LLINFO;   /* flags; can be 0 */

if (sysctl(mib, 6, NULL, &needed, NULL, 0) < 0)
    quit("sysctl error, estimate");

if ( (buf = malloc(needed)) == NULL)
    quit("malloc");

if (sysctl(mib, 6, buf, &needed, NULL, 0) < 0)
    quit("sysctl error, retrieval");

lim = buf + needed;
for (next = buf; next < lim; next += rtm->rtm_msglen) {
    rtm = (struct rt_msghdr *)next;
    ... /* do whatever */
}
```

---

The first three elements in the mib array cause call sysctl\_rtable to process the remaining elements.

mib[4] specifies the operation. Three operations supported.

- 1. NET\_RT\_DUMP: return the routing table corresponding to the address family specified in mib[3]. If the address family is 0, all routes are returned.**

**An RTM\_GET routing message is returned. It contains one routing table entry containing two, three or four socket address structures per message: rt\_key, rt\_gateway, rt\_netmask, and rt\_genmask. The final rt\_gateway might be null.**

- NET\_RT\_FLAGS: the same as the previous call. mib[5] specifies an RTF\_xxx flag ([Figure 18.25](#)). Entries with this flag set are returned.
- NET\_RT\_IFLIST: return information on all the interfaces. If the mib[5] value is nonzero it specifies an interface index and only the interface with the specified if\_index is returned. Otherwise all interfaces on the list are returned.

For each interface one RTM\_IFINFO message is returned with information about the interface itself, followed by one or more RTM\_NEWADDR messages for each ifaddr struct in the interface's if\_addrlist linked list. If the mib[3] value is nonzero, only RTM\_NEWADDR messages are returned for only those interfaces with an address family that matches the mib[3]. Otherwise mib[3] is 0 and information on all active interfaces is returned.

This operation is intended to replace the SIOCGETIFINFO operation ([Figure 4.26](#)).

One problem with this system call is that the amount of information returned can vary, depending on the number of routing table entries or the number of interfaces. The first call to sysctl typically specifies a null pointer for the data argument, which means: don't return any data. The number of bytes of return information. As we saw in [Section 18.2](#), the size of the buffer is determined by the size of the data argument.

19.35, the process then calls malloc, followed by the information. This second call to sysctl again specifies a number of bytes through the fourth argument (which have changed since the previous call), and this time the pointer lim that points just beyond the final byte that was returned. The process then steps through the messages in the buffer, using the rtm\_msglen member to determine the length of each message.

Figure 19.36 shows the values for these six mib[3] entries that various Net/3 programs specify to access the routing table and interface list.

### Figure 19.36. Examples of programs that obtain routing table and interface list

mib[]	arp	route	netstat	routed	gated	rwhod
0	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET	CTL_NET
1	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE	PF_ROUTE
2	0	0	0	0	0	0
3	AF_INET	0	0	AF_INET	0	AF_INET
4	NET_RT_FLAGS	NET_RT_DUMP	NET_RT_DUMP	NET_RT_IFLIST	NET_RT_IFLIST	NET_RT_IFLIST
5	RTF_LLINFO	0	0	0	0	0

The first three programs fetch entries from the routing table and the last three fetch the interface list. The rwhod program supports only the Internet routing protocols, so its mib[3] value of AF\_INET, while gated supports all protocols, so its value for mib[3] is 0.

Figure 19.37 shows the organization of the three programs that obtain the routing table and interface list.

functions that we cover in the following section

**Figure 19.37. Functions that support the sys for routing sockets.**

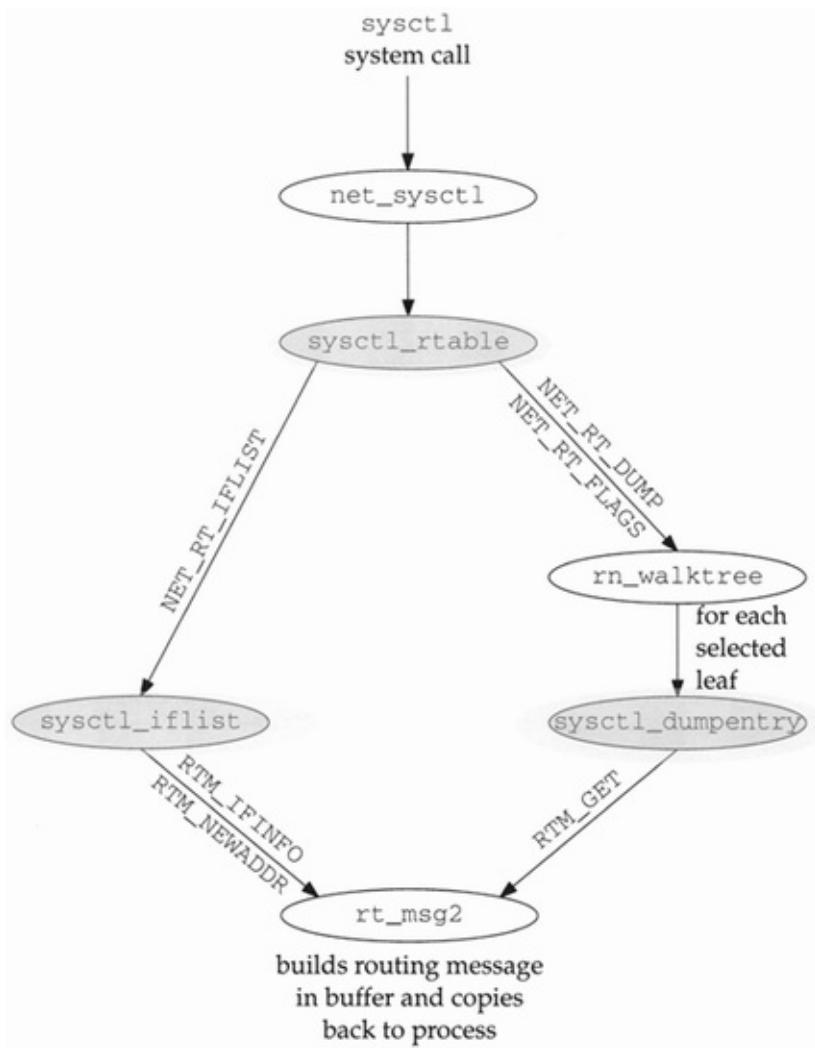


Figure 19.38 shows the `sysctl_rtable` function.

**Figure 19.38. `sysctl_rtable` function: process**

# call requests.

```
-----rtsock.c-----  
705 int  
706 sysctl_rtable(name, namelen, where, given, new, newlen)  
707 int *name;  
708 int namelen;  
709 caddr_t where;  
710 size_t *given;  
711 caddr_t *new;  
712 size_t newlen;  
713 {  
714     struct radix_node_head *rnh;  
715     int i, s, error = EINVAL;  
716     u_char af;  
717     struct walkarg w;  
718     if (new)  
719         return (EPERM);  
  
720     if (namelen != 3)  
721         return (EINVAL);  
722     af = name[0];  
723     Bzero(&w, sizeof(w));  
724     w.w_where = where;  
725     w.w_given = *given;  
726     w.w_needed = 0 - w.w_given;  
727     w.w_op = name[1];  
728     w.w_arg = name[2];  
  
729     s = splnet();  
730     switch (w.w_op) {  
  
731     case NET_RT_DUMP:  
732     case NET_RT_FLAGS:  
733         for (i = 1; i <= AF_MAX; i++)  
734             if ((rnh = rt_tables[i]) && (af == 0 || af == i) &&  
735                 (error = rnh->rnh_walktree(rnh,  
736                                         sysctl_dumpentry, &w)))  
737                 break;  
738         break;  
  
739     case NET_RT_IFLIST:  
740         error = sysctl_iflist(af, &w);  
741     }  
742     splx(s);  
743     if (w.w_tmem)  
744         free(w.w_tmem, M_RTABLE);  
745     w.w_needed += w.w_given;  
746     if (where) {  
747         *given = w.w_where - where;  
748         if (*given < w.w_needed)  
749             return (ENOMEM);  
750     } else {  
751         *given = (11 * w.w_needed) / 10;  
752     }  
753     return (error);  
754 }-----rtsock.c-----
```

## Validate arguments

705-719

The new argument is used when the process is set the value of a variable, which isn't supported by routing tables. Therefore this argument must be

720-721

namelen must be 3 because at this point in the system call, three elements in the name array are name[0], the address family (what the process mib[3]); name[1], the operation (mib[4]); and flags (mib[5]).

## Initialize walkarg structure

723-728

A walkarg structure ([Figure 19.31](#)) is set to 0 and its members are initialized: w\_where is the address of the buffer for the results (this can be null as we mentioned); w\_given is the size of the buffer (this is meaningless on input if w\_where is a null pointer); w\_left must be set on return to the amount of data that has been returned; w\_needed is set to the negative of the buffer size; w\_op is the operation (the NET\_RT\_XXX values); and w\_arg is the flags value.

## Dump routing table

731-738

The NET\_RT\_DUMP and NET\_RT\_FLAGS operations work the same way: a loop is made through all the routing tables (the rt\_tables array), and if the routing table is either the address family argument was 0 or the address family argument matches the family of this routing table, the rn\_walktree function is called to process the entire routing tree. In [Figure 18.17](#) we show that this function is called with three arguments: the address of another function that is called for each node in the routing tree (sysctl\_dumpentry), the third argument is a pointer to anything that rn\_walktree passes to sysctl\_dumpentry function. This argument is a walkarg structure that contains all the information needed for the sysctl call.

## Return interface list

739-740

The NET\_RT\_IFLIST operation calls the function ifnet\_iflist which goes through all the ifnet structures.

## Release buffer

743-744

If a buffer was allocated by rt\_msg2 to contain message, it is now released.

## Update w\_needed

745

The size of each message was added to w\_needed. Since this variable was initialized to the negative its value can now be expressed as

$$w_{needed} = 0 - w_{given} + totalbytes$$

where totalbytes is the sum of all the message by rt\_msg2. By adding the value of w\_given back to w\_needed, we get

$$\begin{aligned} w_{needed} &= 0 - w_{given} + totalbytes \\ &= totalbytes \end{aligned}$$

the total number of bytes. Since the two values in this equation end up canceling each other, where specifies w\_where as a null pointer it need not care about the value of w\_given. Indeed, we see in Figure 19.1 that the variable needed was not initialized.

## **Return actual size of message**

**746-749**

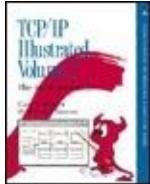
If where is nonnull, the number of bytes stored returned through the given pointer. If this value the size of the buffer specified by the process, it returned because the return information has been

## **Return estimated size of message**

**750-752**

When the where pointer is null, the process just total number of bytes returned. A 10% fudge factor the size, in case the size of the desired tables increased between this call to sysctl and the next.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.15 sysctl\_dumpentry Function

In the previous section we described how this function is called by rn\_walktree, which in turn is called by sysctl\_rtable. Figure 19.39 shows the function.

**Figure 19.39. sysctl\_dumpentry function: process one routing table entry.**

---

```

623 int
624 sysctl_dumpentry(rn, w)
625 struct radix_node *rn;
626 struct walkarg *w;
627 {
628     struct rtentry *rt = (struct rtentry *) rn;
629     int     error = 0, size;
630     struct rt_addrinfo info;

631     if (w->w_op == NET_RT_FLAGS && !(rt->rt_flags & w->w_arg))
632         return 0;
633     bzero((caddr_t) &info, sizeof(info));
634     dst = rt->rt_key(rt);
635     gate = rt->rt_gateway;
636     netmask = rt->rt_mask(rt);
637     genmask = rt->rt_genmask;
638     size = rt_msg2(RTM_GET, &info, 0, w);
639     if (w->w_where && w->w_tmem) {
640         struct rt_msghdr *rtm = (struct rt_msghdr *) w->w_tmem;

641         rtm->rtm_flags = rt->rt_flags;
642         rtm->rtm_use = rt->rt_use;
643         rtm->rtm_rmx = rt->rt_rmx;
644         rtm->rtm_index = rt->rt_ifp->if_index;
645         rtm->rtm_errno = rtm->rtm_pid = rtm->rtm_seq = 0;
646         rtm->rtm_addrs = info.rti_addrs;
647         if (error = copyout((caddr_t) rtm, w->w_where, size))
648             w->w_where = NULL;
649         else
650             w->w_where += size;
651     }
652     return (error);
653 }

```

---

rtsock.c

## 623-630

Each time this function is called, its first argument points to a `radix_node` structure, which is also a pointer to a `rtentry` structure. The second argument points to the `walkarg` structure that was initialized by `sysctl_rtable`.

## Check flags of routing table entry

## 631-632

If the process specified a flag value (mib[5]), this entry is skipped if the `rt_flags` member doesn't have the desired flag set. We see in [Figure 19.36](#) that the `arp` program uses this to select only those entries with the `RTF_LLINFO` flag set, since these are the entries of interest to ARP.

## Form routing message

633-638

The following four pointers in the `rti_info` array are copied from the routing table entry: `dst`, `gate`, `netmask`, and `genmask`. The first two are always nonnull, but the other two can be null. `rt_msg2` forms an `RTM_GET` message.

## Copy message back to process

639-651

If the process wants the message returned and a buffer was allocated by `rt_msg2`, the remainder of the routing message is formed in the buffer pointed to by

w\_tmem and copyout copies the message back to the process. If the copy was successful, w\_where is incremented by the number of bytes copied.

---

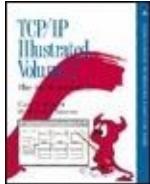
[Team-Fly](#)



[Previous](#)

[Next](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 19. Routing Requests and Routing Messages

### 19.16 sysctl\_iflist Function

This function, shown in [Figure 19.40](#), is called directly by sysctl\_rtable to return the interface list to the process.

**Figure 19.40. sysctl\_iflist function: return list of interfaces and their addresses.**

```

654 int
655 sysctl_iflist(af, w)
656 int      af;
657 struct walkarg *w;
658 {
659     struct ifnet *ifp;
660     struct ifaddr *ifa;
661     struct rt_addrinfo info;
662     int      len, error = 0;
663
664     bzero((caddr_t) & info, sizeof(info));
665     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
666         if (w->w_arg && w->w_arg != ifp->if_index)
667             continue;
668         ifa = ifp->if_addrlist;
669         ifpaddr = ifa->ifa_addr;
670         len = rt_msg2(RTM_IFINFO, &info, (caddr_t) 0, w);
671         ifpaddr = 0;
672         if (w->w_where && w->w_tmem) {
673             struct if_msghdr *ifm;
674
675             ifm = (struct if_msghdr *) w->w_tmem;
676             ifm->ifm_index = ifp->if_index;
677             ifm->ifm_flags = ifp->if_flags;
678             ifm->ifm_data = ifp->if_data;
679             ifm->ifm_addrs = info.rti_addrs;
680             if (error = copyout((caddr_t) ifm, w->w_where, len))
681                 return (error);
682             w->w_where += len;
683         }
684         while (ifa = ifa->ifa_next) {
685             if (af && af != ifa->ifa_addr->sa_family)
686                 continue;
687             ifaaddr = ifa->ifa_addr;
688             netmask = ifa->ifa_netmask;
689             brdaddr = ifa->ifa_dstaddr;
690             len = rt_msg2(RTM_NEWWADDR, &info, 0, w);
691             if (w->w_where && w->w_tmem) {
692                 struct ifa_msghdr *ifam;
693
694                 ifam = (struct ifa_msghdr *) w->w_tmem;
695                 ifam->ifam_index = ifa->ifa_ifp->if_index;
696                 ifam->ifam_flags = ifa->ifa_flags;
697                 ifam->ifam_metric = ifa->ifa_metric;
698                 ifam->ifam_addrs = info.rti_addrs;
699                 if (error = copyout(w->w_tmem, w->w_where, len))
700                     return (error);
701                 w->w_where += len;
702             }
703         }
704     }

```

This function is a for loop that iterates through each interface starting with the one pointed to by ifnet. Then a while loop proceeds through the linked list of ifaddr structures for each interface. An

RTM\_IFINFO routing message is generated for each interface and an RTM\_NEWADDR message for each address.

## Check interface index

654-666

The process can specify a nonzero flags argument (mib[5] in [Figure 19.36](#)) to select only the interface with a matching if\_index value.

## Build routing message

667-670

The only socket address structure returned with the RTM\_IFINFO message is ifpaddr. The message is built by rt\_msg2. The pointer ifpaddr in the info structure is then set to 0, since the same info structure is used for generating the subsequent RTM\_NEWADDR messages.

## Copy message back to process

671-681

If the process wants the message returned, the remainder of the if\_msghdr structure is filled in, copyout copies the buffer to the process, and w\_where is incremented.

### Iterate through address structures, check address family

682-684

Each ifaddr structure for the interface is processed and the process can specify a nonzero address family (mib[3] in [Figure 19.36](#)) to select only the interface addresses of the given family.

### Build routing message

685-688

Up to three socket address structures are returned in each RTM\_NEWADDR message: ifaaddr, netmask, and brdaddr. The message is built by rt\_msg2.

## Copy message back to process

689-699

If the process wants the message returned, the remainder of the ifa\_msghdr structure is filled in, copyout copies the buffer to the process, and w\_where is incremented.

701

These three pointers in the info array are set to 0, since the same array is used for the next interface message.



## Chapter 19. Routing Requests and Routing Tables

---

### 19.17 Summary

Routing messages all have the same format after the header, which contains a fixed number of socket address structures. There are four types of routing messages, corresponding to a different fixed-length structure. The first structure identifies the length, version, and type of the message. The second structure identifies which socket address structures follow.

These messages are passed between a process and a routing daemon. Messages can be passed in either direction, one at a time, through a single socket. This allows a superuser process complete access to the kernel's routing tables. This is how routing daemons such as ipfw(4) implement their desired routing policy.

Alternatively any process can read the contents of the kernel's routing tables via the getrtable system call. This does not involve a routing socket. The entire result, normally consisting of many individual routing entries, is returned in a single system call. Since the process does not know the size of the result, it must allocate memory for the system call to return this size without returning an error.

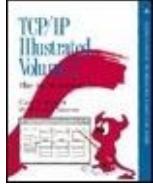
## Exercises

**19.1** What is the difference in the RTF\_DYNAM] for a given routing table entry?

What happens when the default route is e

**19.2** bsdi \$ **route add default -c**

**19.3** Estimate the space required by sysctl to c entries and 20 routes.



[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Chapter 20. Routing Sockets

[Section 20.1. Introduction](#)

[Section 20.2. routedomain and protosw Structures](#)

[Section 20.3. Routing Control Blocks](#)

[Section 20.4. raw\\_init Function](#)

[Section 20.5. route\\_output Function](#)

[Section 20.6. rt\\_xaddrs Function](#)

[Section 20.7. rt\\_setmetrics Function](#)

[Section 20.8. raw\\_input Function](#)

[Section 20.9. route\\_usrreq Function](#)

[Section 20.10. raw\\_usrreq Function](#)

[Section 20.11. raw\\_attach,  
raw\\_detach, and raw\\_disconnect  
Functions](#)

## Section 20.12. Summary

---

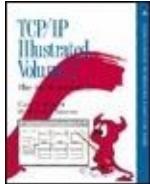
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.1 Introduction

A process sends and receives the routing messages described in the previous chapter by using a socket in the *routing domain*. The socket system call is issued specifying a family of PF\_ROUTE and a socket type of SOCK\_RAW.

The process can then send five routing messages to the kernel:

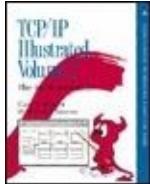
#### **1. RTM\_ADD: add a new route.**

- RTM\_DELETE: delete an existing route.
- RTM\_GET: fetch all the information about a route.

- RTM\_CHANGE: change the gateway, interface, or metrics of an existing route.
- RTM\_LOCK: specify which metrics the kernel should not modify.

Additionally, the process can receive any of the other seven types of routing messages that are generated by the kernel when some event, such as interface down, redirect received, etc., occurs.

This chapter looks at the routing domain, the routing control blocks that are created for each routing socket, the function that handles messages from a process (`route_output`), the function that sends routing messages to one or more processes (`raw_input`), and the various functions that support all the socket operations on a routing socket.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

## 20.2 routedomain and protosw Structures

Before describing the routing socket functions, we need to discuss additional details about the routing domain; the SOCK\_RAW protocol supported in the routing domain; and routing control blocks, one of which is associated with each routing socket.

[Figure 20.1](#) lists the domain structure for the PF\_ROUTE domain, named `routedomain`.

**Figure 20.1. routedomain structure.**

Member	Value	Description
dom_family	<code>PF_ROUTE</code>	protocol family for domain name
dom_name	<code>route</code>	
dom_init	<code>route_init</code>	domain initialization, Figure 18.30
dom_externalize	0	not used in routing domain
dom_dispose	0	not used in routing domain
dom_protosw	<code>routesw</code>	protocol switch structure, Figure 20.2
dom_protoswNPROTOSW		pointer past end of protocol switch structure
dom_next		filled in by <code>domaininit</code> , Figure 7.15
dom_rtattach	0	not used in routing domain
dom_rtoffset	0	not used in routing domain
dom_maxrtkey	0	not used in routing domain

Unlike the Internet domain, which supports multiple protocols (TCP, UDP, ICMP, etc.), only one protocol (of type `SOCK_RAW`) is supported in the routing domain. [Figure 20.2](#) lists the protocol switch entry for the `PF_ROUTE` domain.

## Figure 20.2. The routing protocol protosw structure.

Member	<code>routesw[0]</code>	Description
<code>pr_type</code>	<code>SOCK_RAW</code>	raw socket
<code>pr_domain</code>	<code>&amp;routedomain</code>	part of the routing domain
<code>pr_protocol</code>	0	
<code>pr_flags</code>	<code>PR_ATOMIC PR_ADDR</code>	socket layer flags, not used by protocol processing
<code>pr_input</code>	<code>raw_input</code>	this entry not used; <code>raw_input</code> called directly
<code>pr_output</code>	<code>route_output</code>	called for <code>PRU_SEND</code> requests
<code>pr_ctlinput</code>	<code>raw_ctlinput</code>	control input function
<code>pr_ctloutput</code>	0	not used
<code>pr_usrreq</code>	<code>route_usrreq</code>	respond to communication requests from a process
<code>pr_init</code>	<code>raw_init</code>	initialization
<code>pr_fasttimo</code>	0	not used
<code>pr_slowtimo</code>	0	not used
<code>pr_drain</code>	0	not used
<code>pr_sysctl</code>	<code>sysctl_rtable</code>	for <code>sysctl(8)</code> system call

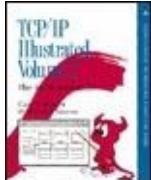
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.3 Routing Control Blocks

Each time a routing socket is created with a call of the form

```
socket(PF_ROUTE, SOCK_RAW, protocol);
```

the corresponding PRU\_ATTACH request to the protocol's user-request function (route\_usrreq) allocates a routing control block and links it to the socket structure. The *protocol* can restrict the messages sent to the process on this socket to a particular family. If a *protocol* of AF\_INET is specified, for example, only routing messages containing Internet addresses will be sent to the process. A *protocol* of 0 causes all routing messages from the kernel to be sent on the socket.

Recall that we call these structures *routing control blocks*.

*blocks*, not *raw control blocks*, to avoid confusion with the raw IP control blocks in [Chapter 32](#).

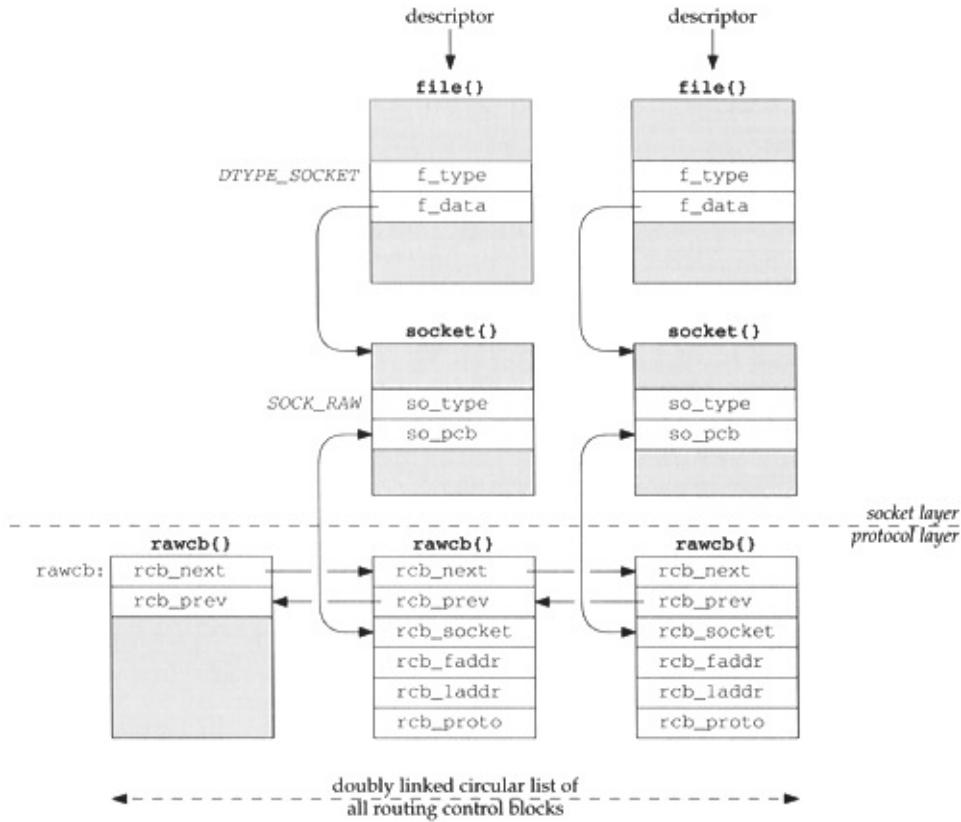
[Figure 20.3](#) shows the definition of the rawcb structure.

### Figure 20.3. rawcb structure.

```
39 struct rawcb {  
40     struct rawcb *rcb_next; /* doubly linked list */  
41     struct rawcb *rcb_prev;  
42     struct socket *rcb_socket; /* back pointer to socket */  
43     struct sockaddr *rcb_faddr; /* destination address */  
44     struct sockaddr *rcb_laddr; /* socket's address */  
45     struct sockproto rcb_proto; /* protocol family, protocol */  
46 };  
47 #define sotorawcb(so) ((struct rawcb *)(so)->so_pcb)  
----- raw_cb.h -----
```

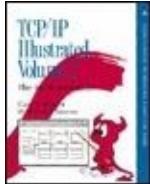
Additionally, a global of the same name, rawcb, allocated as the head of the doubly linked list. [Figure 20.4](#) shows the arrangement.

### Figure 20.4. Relationship of raw protocol control blocks to other data structures.



39-47

We showed the sockproto structure in [Figure 19](#). Its sp\_family member is set to PF\_ROUTE and its sp\_protocol member is set to the third argument of the socket system call. The rcb\_faddr member is permanently set to point to route\_src, which was described with [Figure 19.26](#). rcb\_laddr is always null pointer.



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.4 raw\_init Function

The raw\_init function, shown in [Figure 20.5](#), is the protocol initialization function in the protosw structure in [Figure 20.2](#). We described the entire initialization of the routing domain with [Figure 18.29](#).

**Figure 20.5. raw\_init function: initialize doubly linked list of routing control blocks.**

```
38 void  
39 raw_init()  
40 {  
41     rawcb.rcb_next = rawcb.rcb_prev = &rawcb;  
42 }
```

The function initializes the doubly linked list of routing control blocks by setting the next and previous pointers of the head structure to point to itself.

---

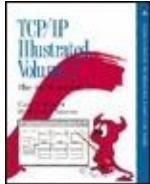
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

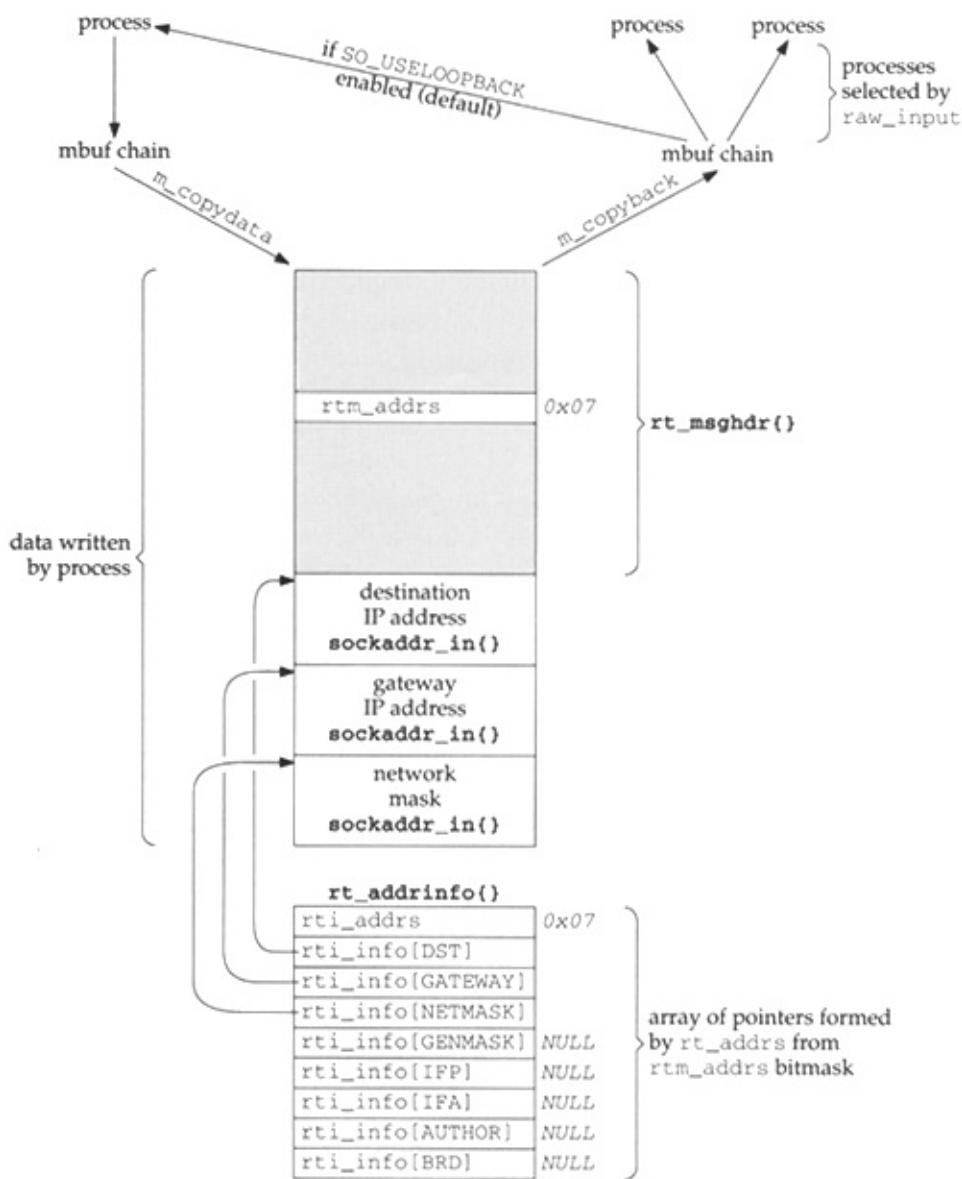
### 20.5 route\_output Function

As we showed in [Figure 18.11](#), `route_output` is called when the `PRU_SEND` request is issued to the protocol's user-request function, which is the result of a write operation by a process to a routing socket. In [Figure 18.9](#) we indicated that five different types of routing messages are accepted by the kernel from a process.

Since this function is invoked as a result of a write by a process, the data from the process (the routing message to process) is in an mbuf chain from `sosend`. [Figure 20.6](#) shows an overview of the processing steps, assuming the process sends an `RTM_ADD` command, specifying three

addresses: the destination, its gateway, and a network mask (hence this is a network route, not a host route).

**Figure 20.6. Example processing of an RTM\_ADD command from a process.**



There are numerous points to note in this figure, most of which we'll cover as we proceed through the source code for `route_output`. Also note that, to save space, we omit the `RTAX_` prefix for each array index in the `rt_addrinfo` structure.

- The process specifies which socket address structures follow the fixed-length `rt_msghdr` structure by setting the bitmask `rtm_addrs`. We show a bitmask of `0x07`, which corresponds to a destination address, a gateway address, and a network mask ([Figure 19.19](#)). The `RTM_ADD` command requires the first two; the third is optional. Another optional address, the `genmask` specifies the mask to be used for generating cloned routes.
- The write system call (the `sosend` function) copies the buffer from the process into an mbuf chain in the kernel.
- `m_copydata` copies the mbuf chain into a buffer that `route_output` obtains using `malloc`. It is easier to access all the

information in the structure and the socket address structures that follow when stored in a single contiguous buffer than it is when stored in an mbuf chain.

- The function `rt_xaddrs` is called by `route_output` to take the bitmask and build the `rt_addrinfo` structure that points into the buffer. The code in `route_output` references these structures using the names shown in the fifth column in [Figure 19.19](#). The bitmask is also copied into the `rti_addrs` member.
- `route_output` normally modifies the `rt_msghdr` structure. If an error occurs, the corresponding `errno` value is returned in `rtm_errno` (for example, `EEXIST` if the route already exists); otherwise the flag `RTF_DONE` is logically ORed into the `rtm_flags` supplied by the process.
- The `rt_msghdr` structure and the addresses that follow become input to 0 or more processes that are reading

from a routing socket. The buffer is first converted back into an mbuf chain by `m_copyback`. `raw_input` goes through all the routing PCBs and passes a copy to the appropriate processes. We also show that a process with a routing socket receives a copy of each message it writes to that socket unless it disables the `SO_USELOOPBACK` socket option.

To avoid receiving a copy of their own routing messages, some programs, such as `route`, call `shutdown` with a second argument of 0 to prevent any data from being received on the routing socket.

We examine the source code for `route_output` in seven parts. [Figure 20.7](#) shows an overview of the function.

**Figure 20.7. Summary of `route_output` processing steps.**

---

```
int
route_output()
{
    R_Malloc() to allocate buffer;
    m_copydata() to copy from mbuf chain into buffer;
    rt_xaddrs() to build rt_addrinfo();

    switch (message type) {
    case RTM_ADD:
        rtrequest(RTM_ADD);
        rt_setmetrics();
        break;

    case RTM_DELETE:
        rtrequest(RTM_DELETE);
        break;

    case RTM_GET:
    case RTM_CHANGE:
    case RTM_LOCK:
        rtalloc1();

        switch (message type) {
        case RTM_GET:
            rt_msg2(RTM_GET);
            break;

        case RTM_CHANGE:
            change appropriate fields;
            /* fall through */

        case RTM_LOCK:
            set rmx_locks;
            break;
        }
        break;
    }

    set rtm_error if error, else set RTF_DONE flag;
    m_copyback() to copy from buffer into mbuf chain;
    raw_input(); /* mbuf chain to appropriate processes */
}
```

---

The first part of route\_output is shown in Figure 20.8.

**Figure 20.8. route\_output function: initial processing, copy message from mbuf chain.**

```

113 int
114 route_output(m, so)
115 struct mbuf *m;
116 struct socket *so;
117 {
118     struct rt_msghdr *rtm = 0;
119     struct rtentry *rt = 0;
120     struct rtentry *saved_rnt = 0;
121     struct rt_addrinfo info;
122     int len, error = 0;
123     struct ifnet *ifp = 0;
124     struct ifaddr *ifa = 0;
125 #define senderr(e) { error = e; goto flush; }
126     if (m == 0 || ((m->m_len < sizeof(long)) &&
127                     (m = m_pullup(m, sizeof(long))) == 0))
128         return (ENOBUFS);
129     if ((m->m_flags & M_PKTHDR) == 0)
130         panic("route_output");
131     len = m->m_pkthdr.len;
132     if (len < sizeof(*rtm) ||
133         len != mtod(m, struct rt_msghdr *)->rtm_msflen) {
134         dst = 0;
135         senderr(EINVAL);
136     }
137     R_Malloc(rtm, struct rt_msghdr *, len);
138     if (rtm == 0) {
139         dst = 0;
140         senderr(ENOBUFS);
141     }
142     m_copydata(m, 0, len, (caddr_t) rtm);
143     if (rtm->rtm_version != RTM_VERSION) {
144         dst = 0;
145         senderr(EPROTONOSUPPORT);
146     }
147     rtm->rtm_pid = curproc->p_pid;
148     info.rti_addrs = rtm->rtm_addrs;
149     rt_xaddrs((caddr_t) (rtm + 1), len + (caddr_t) rtm, &info);
150     if (dst == 0)
151         senderr(EINVAL);
152     if (genmask) {
153         struct radix_node *t;
154         t = rn_addmask((caddr_t) genmask, 1, 2);
155         if (t && Bcmp(genmask, t->rn_key, *(u_char *) genmask) == 0)
156             genmask = (struct sockaddr *) (t->rn_key);
157         else
158             senderr(ENOBUFS);
159     }

```

rtsock.c

## Check mbuf for validity

113-136

The mbuf chain is checked for validity: its length must be at least the size of an

`rt_msghdr` structure. The first longword is fetched from the data portion of the mbuf, which contains the `rtm_msglen` value.

## Allocate buffer

`137-142`

A buffer is allocated to hold the entire message and `m_copydata` copies the message from the mbuf chain into the buffer.

## Check version number

`143-146`

The version of the message is checked. In the future, should a new version of the routing messages be introduced, this member could be used to provide support for older versions.

`147-149`

The process ID is copied into `rtm_pid` and the bitmask supplied by the process is

copied into info.rti\_addrs, a structure local to this function. The function rt\_xaddrs (shown in the next section) fills in the eight socket address pointers in the info structure to point into the buffer now containing the message.

## Destination address required

150-151

A destination address is a required address for all commands. If the info.rti\_info[RTAX\_DST] element is a null pointer, EINVAL is returned. Remember that dst refers to this array element ([Figure 19.19](#)).

## Handle optional genmask

152-159

A genmask is optional and is used as the network mask for routes created when the RTF\_CLONING flag is set ([Figure 19.8](#)). rn\_addmask adds the mask to the tree of masks, first searching for an existing entry

for the mask and then referencing that entry if found. If the mask is found or added to the mask tree, an additional check is made that the entry in the mask tree really equals the genmask value, and, if so, the genmask pointer is replaced with a pointer to the mask in the mask tree.

[Figure 20.9](#) shows the next part of `route_output`, which handles the `RTM_ADD` and `RTM_DELETE` commands.

## Figure 20.9. `route_output` function: process `RTM_ADD` and `RTM_DELETE` commands.

```
160     switch (rtm->rtm_type) {                                     rtssock.c
161         case RTM_ADD:
162             if (gate == 0)
163                 senderr(EINVAL);
164             error = rtrequest(RTM_ADD, dst, gate, netmask,
165                                 rtm->rtm_flags, &saved_nrt);
166             if (error == 0 && saved_nrt) {
167                 rt_setmetrics(rtm->rtm_inits,
168                               &rtm->rtm_rmx, &saved_nrt->rt_rmx);
169                 saved_nrt->rt_refcnt--;
170                 saved_nrt->rt_genmask = genmask;
171             }
172             break;
173         case RTM_DELETE:
174             error = rtrequest(RTM_DELETE, dst, gate, netmask,
175                               rtm->rtm_flags, (struct rtentry **) 0);
176             break;

```

162-163

An RTM\_ADD command requires the process to specify a gateway.

164-165

rtrequest processes the request. The netmask pointer can be null if the route being entered is a host route. If all is OK, the pointer to the new routing table entry is returned through saved\_nrt.

166-172

The rt\_metrics structure is copied from the caller's buffer into the routing table entry. The reference count is decremented and the genmask pointer is stored (possibly a null pointer).

173-176

Processing the RTM\_DELETE command is simple because all the work is done by rtrequest. Since the final argument is a null pointer, rtrequest calls rtfree if the reference count is 0, deleting the entry from the routing table ([Figure 19.7](#)).

The next part of the processing is shown in

[Figure 20.10](#), which handles the common code for the RTM\_GET, RTM\_CHANGE, and RTM\_LOCK commands.

## Figure 20.10. route\_output function: common processing for RTM\_GET, RTM\_CHANGE, and RTM\_LOCK.

```
----- rtsock.c
177 case RTM_GET:
178 case RTM_CHANGE:
179 case RTM_LOCK:
180     rt = rtalloc1(dst, 0);
181     if (rt == 0)
182         . senderr(ESRCH);
183     if (rtm->rtm_type != RTM_GET) {      /* XXX: too grotty */
184         struct radix_node *rn;
185         extern struct radix_node_head *mask_rnhead;
186
187         if (Bcmp(dst, rt_key(rt), dst->sa_len) != 0)
188             senderr(ESRCH);
189         if (netmask && (rn = rn_search(netmask,
190                         mask_rnhead->rnh_treetop)))
191             netmask = (struct sockaddr *) rn->rn_key;
192         for (rn = rt->rt_nodes; rn; rn = rn->rn_dupedkey)
193             if (netmask == (struct sockaddr *) rn->rn_mask)
194                 break;
195         if (rn == 0)
196             senderr(ETOOMANYREFS);
197         rt = (struct rtentry *) rn;
198     }
----- rtsock.c
```

## Locate existing entry

177-182

Since all three commands reference an existing entry, rtalloc1 locates the entry. If the entry isn't found, ESRCH is returned.

## Do not allow network match

183-187

For the RTM\_CHANGE and RTM\_LOCK commands, a network match is inadequate: an exact match with the routing table key is required. Therefore, if the dst argument doesn't equal the routing table key, the match was a network match and ESRCH is returned.

## Use network mask to find correct entry

188-193

Even with an exact match, if there are duplicate keys, each with a different network mask, the correct entry must still be located. If a netmask argument was supplied, it is looked up in the mask table (mask\_rnhead). If found, the netmask pointer is replaced with the pointer to the mask in the mask tree. Each leaf node in the duplicate key list is examined, looking for an entry with an rn\_mask pointer that equals netmask. This test compares the

pointers, not the structures that they point to. This works because all masks appear in the mask tree, and only one copy of each unique mask is stored in this tree. In the common case, keys are not duplicated, so the for loop iterates once. If a host entry is being modified, a mask must not be specified and then both netmask and rn\_mask are null pointers (which are equal). But if an entry that has an associated mask is being modified, that mask must be specified as the netmask argument.

## 194-195

If the for loop terminates without finding a matching network mask, ETOOMANYREFS is returned.

The comment XXX is because this function must go to all this work to find the desired entry. All these details should be hidden in another function similar to rtalloc1 that detects a network match and handles a mask argument.

The next part of this function, shown in [Figure 20.11](#), continues processing the RTM\_GET command. This command is unique among the commands supported by route\_output in that it can return more data than it was passed. For example, only a single socket address structure is required as input, the destination, but at least two are returned: the destination and its gateway. With regard to [Figure 20.6](#), this means the buffer allocated for m\_copydata to copy into might need to be increased in size.

## **Figure 20.11. route\_output function: RTM\_GET processing.**

---

```

198     switch (rtm->rtm_type) {
199         case RTM_GET:
200             dst = rt->rt_key(rt);
201             gate = rt->rt_gateway;
202             netmask = rt->rt_mask(rt);
203             genmask = rt->rt_genmask;
204             if (rtm->rtm_addrs & (RTA_IFF | RTA_IFA)) {
205                 if (ifp = rt->rt_ifp) {
206                     ifpaddr = ifp->if_addrlist->ifa_addr;
207                     ifaaddr = rt->rt_ifa->ifa_addr;
208                     rtm->rtm_index = ifp->if_index;
209                 } else {
210                     ifpaddr = 0;
211                     ifaaddr = 0;
212                 }
213             }
214             len = rt_msg2(RTM_GET, &info, (caddr_t) 0,
215                           (struct walkarg *) 0);
216             if (len > rtm->rtm_msflen) {
217                 struct rt_msghdr *new_rtm;
218                 R_Malloc(new_rtm, struct rt_msghdr *, len);
219                 if (new_rtm == 0)
220                     senderr(ENOBUFS);
221                 Bcopy(rtm, new_rtm, rtm->rtm_msflen);
222                 Free(rtm);
223                 rtm = new_rtm;
224             }
225             (void) rt_msg2(RTM_GET, &info, (caddr_t) rtm,
226                           (struct walkarg *) 0);
227             rtm->rtm_flags = rt->rt_flags;
228             rtm->rtm_rmx = rt->rt_rmx;
229             rtm->rtm_addrs = info.rti_addrs;
230             break;

```

---

rtsock.c

## Return destination, gateway, and masks

198-203

Four pointers are stored in the `rti_info` array: `dst`, `gate`, `netmask`, and `genmask`. The latter two might be null pointers. These pointers in the `info` structure point to the socket address structures that will be returned to the process.

## Return interface information

204-213

The process can set the masks RTA\_IFP and RTA\_IFA in the rtm\_flags bitmask. If either or both are set, the process wants to receive the contents of both the ifaddr structures pointed to by this routing table entry: the link-level address of the interface (pointed to by rt\_ifp->if\_addrlist) and the protocol address for this entry (pointed to by rt\_ifa->ifa\_addr). The interface index is also returned.

## Construct reply

214-224

rt\_msg2 is called with a null third pointer to calculate the length of the routing message corresponding to RTM\_GET and the addresses pointed to by the info structure. If the length of the result message exceeds the length of the input message, then a new buffer is allocated, the input message is copied into the new

buffer, the old buffer is released, and rtm is set to point to the new buffer.

225-230

rt\_msg2 is called again, this time with a nonnull third pointer, which builds the result message in the buffer. The final three members in the rt\_msghdr structure are then filled in.

[Figure 20.12](#) shows the processing of the RTM\_CHANGE and RTM\_LOCK commands.

**Figure 20.12. route\_output function:  
RTM\_CHANGE and RTM\_LOCK  
processing.**

```

231     case RTM_CHANGE:
232         if (gate && rt_setgate(rt, rt_key(rt), gate))
233             senderr(EDQUOT);
234         /* new gateway could require new ifaddr, ifp; flags may also be
235            different: ifp may be specified by ll sockaddr when protocol
236            address is ambiguous */
237         if (ifpaddr && (ifa = ifa_ifwithnet(ifpaddr)) &&
238             (ifp = ifa->ifa_ifp))
239             ifa = ifaof_ifpforaddr(ifaaddr ? ifaaddr : gate,
240                                   ifp);
241         else if ((ifaaddr && (ifa = ifa_ifwithaddr(ifaaddr))) ||
242                  (ifa = ifa_ifwithroute(rt->rt_flags,
243                                         rt_key(rt), gate)))
244             ifp = ifa->ifa_ifp;
245         if (ifa) {
246             struct ifaddr *oifa = rt->rt_ifa;
247             if (oifa != ifa) {
248                 if (oifa && oifa->ifa_rtrequest)
249                     oifa->ifa_rtrequest(RTM_DELETE,
250                                           rt, gate);
251                 IFAFREE(rt->rt_ifa);
252                 rt->rt_ifa = ifa;
253                 ifa->ifa_refcnt++;
254                 rt->rt_ifp = ifp;
255             }
256             rt_setmetrics(rtm->rtm_inits, &rtm->rtm_rmx,
257                           &rt->rt_rmx);
258             if (rt->rt_ifa && rt->rt_ifa->ifa_rtrequest)
259                 rt->rt_ifa->ifa_rtrequest(RTM_ADD, rt, gate);
260             if (genmask)
261                 rt->rt_genmask = genmask;
262             /*
263              * Fall into
264              */
265         case RTM_LOCK:
266             rt->rt_rmx.rmx_locks &= ~(rtm->rtm_inits);
267             rt->rt_rmx.rmx_locks |=
268                 (rtm->rtm_inits & rtm->rtm_rmx.rmx_locks);
269             break;
270         }
271     }
272     break;
273 default:
274     senderr(EOPNOTSUPP);
275 }
```

rtsock.c

## Change gateway

231-233

If a gate address was passed by the process, `rt_setgate` is called to change the gateway for the entry.

## Locate new interface

234-244

The new gateway (if changed) can also require new `rt_ifp` and `rt_ifa` pointers. The process can specify these new values by passing either an `ifpaddr` socket address structure or an `ifaaddr` socket address structure. The former is tried first, and then the latter. If neither is passed by the process, the `rt_ifp` and `rt_ifa` pointers are left alone.

## Check if interface changed

245-256

If an interface was located (`ifa` is nonnull), then the existing `rt_ifa` pointer for the route is compared to the new value. If it has changed, new values for `rt_ifp` and `rt_ifa` are stored in the routing table entry. Before doing this the interface request function (if defined) is called with a command of `RTM_DELETE`. The delete is required because the link-layer information

from one type of network to another can be quite different, say changing a route from an X.25 network to an Ethernet, and the output routines must be notified.

## Update metrics

257-258

The metrics in the routing table entry are updated by `rt_setmetrics`.

## Call interface request function

259-260

If an interface request function is defined, it is called with a command of `RTM_ADD`.

## Store clone generation mask

261-262

If the process specifies the genmask argument, the pointer to the mask that was obtained in [Figure 20.8](#) is saved in

`rt_genmask`.

## Update bitmask of locked metrics

266-270

The RTM\_LOCK command updates the bitmask stored in `rt_rmx.rmx_locks`.

[Figure 20.13](#) shows the values of the different bits in this bitmask, one value per metric.

**Figure 20.13. Constants to initialize or lock metrics.**

Constant	Value	Description
<code>RTV_MTU</code>	0x01	initialize or lock <code>rmx_mtu</code>
<code>RTV_HOPCOUNT</code>	0x02	initialize or lock <code>rmx_hopcount</code>
<code>RTV_EXPIRE</code>	0x04	initialize or lock <code>rmx_expire</code>
<code>RTV_RPIPE</code>	0x08	initialize or lock <code>rmx_recvpipe</code>
<code>RTV_SPIPE</code>	0x10	initialize or lock <code>rmx_sendpipe</code>
<code>RTV_SSTHRESH</code>	0x20	initialize or lock <code>rmx_ssthresh</code>
<code>RTV_RTT</code>	0x40	initialize or lock <code>rmx_rtt</code>
<code>RTV_RTTVAR</code>	0x80	initialize or lock <code>rmx_rttvar</code>

The `rmx_locks` member of the `rt_metrics` structure in the routing table entry is the bitmask telling the kernel which metrics to leave alone. That is, those metrics specified by `rmx_locks` won't be updated

by the kernel. The only use of these metrics by the kernel is with TCP, as noted with [Figure 27.3](#). The rmx\_pktsent metric cannot be locked or initialized, but it turns out this member is never even referenced or updated by the kernel.

The rtm\_inits value in the message from the process specifies the bitmask of which metrics were just initialized by rt\_setmetrics. The rtm\_rmx.rmx\_locks value in the message specifies the bitmask of which metrics should now be locked. The value of rt\_rmx.rmx\_locks is the bitmask in the routing table of which metrics are currently locked. First, any bits to be initialized (rtm\_inits) are unlocked. Any bits that are both initialized (rtm\_inits) and locked (rtm\_rmx.rmx\_locks) are locked.

273-275

This default is for the switch at the beginning of [Figure 20.9](#) and catches any of the routing commands other than the five that are supported in messages from a process.

The final part of route\_output, shown in Figure 20.14, sends the reply to raw\_input.

## Figure 20.14. route\_output function: pass results to raw\_input.

```
-----rtsock.c-----  
276     flush:  
277     if (rtm) {  
278         if (error)  
279             .rtm->rtm_errno = error;  
280         else  
281             rtm->rtm_flags |= RTF_DONE;  
282     }  
283     if (rt)  
284         rtfree(rt);  
285     {  
286         struct rawcb *rp = 0;  
287         /*  
288          * Check to see if we don't want our own messages.  
289          */  
290         if ((so->so_options & SO_USELOOPBACK) == 0) {  
291             if (route_cb.any_count <= 1) {  
292                 if (rtm)  
293                     Free(rtm);  
294                 m_freem(m);  
295                 return (error);  
296             }  
297             /* There is another listener, so construct message */  
298             rp = sotorawcb(so);  
299         }  
300         if (rtm) {  
301             m_copyback(m, 0, rtm->rtm_msghlen, (caddr_t) rtm);  
302             Free(rtm);  
303         }  
304         if (rp)  
305             rp->rcb_proto.sp_family = 0; /* Avoid us */  
306         if (dst)  
307             route_proto.sp_protocol = dst->sa_family;  
308             raw_input(m, &route_proto, &route_src, &route_dst);  
309         if (rp)  
310             rp->rcb_proto.sp_family = PF_ROUTE;  
311     }  
312     return (error);  
313 }-----rtsock.c-----
```

Return error or OK

276-282

flush is the label jumped to by the senderr macro defined at the beginning of the function. If an error occurred it is returned in the rtm\_errno member; otherwise the RTF\_DONE flag is set.

## Release held route

283-284

If a route is being held, it is released. The call to rtalloc1 at the beginning of [Figure 20.10](#) holds the route, if found.

## No process to receive message

285-296

The SO\_USELOOPBACK socket option is true by default and specifies that the sending process is to receive a copy of each routing message that it writes to a routing socket. (If the sender doesn't receive a copy, it can't receive any of the information returned by RTM\_GET.) If that

option is not set, and the total count of routing sockets is less than or equal to 1, there are no other processes to receive the message and the sender doesn't want a copy. The buffer and mbuf chain are both released and the function returns.

## **Other listeners but no loopback copy**

297-299

There is at least one other listener but the sending process does not want a copy. The pointer rp, which defaults to null, is set to point to the routing control block for the sender and is also used as a flag that the sender doesn't want a copy.

## **Convert buffer into mbuf chain**

300-303

The buffer is converted back into an mbuf chain ([Figure 20.6](#)) and the buffer released.

## Avoid loopback copy

304-305

If rp is set, some other process might want the message but the sender does not want a copy. The sp\_family member of the sender's routing control block is temporarily set to 0, but the sp\_family of the message (the route\_proto structure, shown with [Figure 19.26](#)) has a family of PF\_ROUTE. This trick prevents raw\_input from passing a copy of the result to the sending process because raw\_input does not pass a copy to any socket with an sp\_family of 0.

## Set address family of routing message

306-308

If dst is a nonnull pointer, the address family of that socket address structure becomes the protocol of the routing message. With the Internet protocols this value would be PF\_INET. A copy is passed to the appropriate listeners by raw\_input.

309-313

If the sp\_family member in the calling process was temporarily set to 0, it is reset to PF\_ROUTE, its normal value.

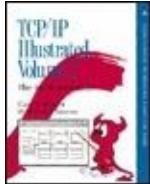
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.6 rt\_xaddrs Function

The `rt_xaddrs` function is called only once from `route_output` (Figure 20.8) after the routing message from the process has been copied from the mbuf chain into a buffer and after the bitmask from the process (`rtm_addrs`) has been copied into the `rti_info` member of an `rt_addrinfo` structure. The purpose of `rt_xaddrs` is to take this bitmask and set the pointers in the `rti_info` array to point to the corresponding address in the buffer. Figure 20.15 shows the function.

**Figure 20.15. `rt_xaddrs` function: fill `rti_info` array with pointers.**

```
-----rtsock.c
330 #define ROUNDUP(a) \
331     ((a) > 0 ? (1 + (((a) - 1) | (sizeof(long) - 1))) : sizeof(long))
332 #define ADVANCE(x, n) (x += ROUNDUP((n)->sa_len))

333 static void
334 rt_xaddrs(cp, cplim, rtinfo)
335 caddr_t cp, cplim;
336 struct rt_addrinfo *rtinfo;
337 {
338     struct sockaddr *sa;
339     int i;
340     bzero(rtinfo->rti_info, sizeof(rtinfo->rti_info));
341     for (i = 0; (i < RTAX_MAX) && (cp < cplim); i++) {
342         if ((rtinfo->rti_addrs & (1 << i)) == 0)
343             continue;
344         rtinfo->rti_info[i] = sa = (struct sockaddr *) cp;
345         ADVANCE(cp, sa);
346     }
347 }
```

---

-----rtsock.c

## 330-340

The array of pointers is set to 0 so all the pointers to address structures not appearing in the bitmask will be null.

## 341-347

Each of the 8 (RTAX\_MAX) possible bits in the bitmask is tested and, if set, a pointer is stored in the rti\_info array to the corresponding socket address structure. The ADVANCE macro takes the sa\_len field of the socket address structure, rounds it up to the next multiple of 4 bytes, and increments the pointer cp accordingly.

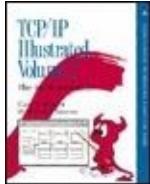
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.7 `rt_setmetrics` Function

This function was called twice from `route_output`: when a new route was added and when an existing route was changed. The `rtm_inits` member in the routing message from the process specifies which of the metrics the process wants to initialize from the `rtm_rmx` array. The bit values in the bitmask are shown in Figure 20.13.

Notice that both `rtm_addrs` and `rtm_inits` are bitmasks in the message from the process, the former specifying the socket address structures that follow, and the latter specifying which metrics are to be initialized. Socket address structures

whose bits don't appear in rtm\_addrs don't even appear in the routing message, to save space. But the entire rt\_metrics array always appears in the fixed-length rt\_msghdr structure elements in the array whose bits are not set in rtm\_inits are ignored.

[Figure 20.16](#) shows the rt\_setmetrics function.

## Figure 20.16. rt\_setmetrics function: set elements of the rt\_metrics structure.

```
314 void rt_setmetrics(which, in, out) ---rtsock.c
315 rt_setmetrics(which, in, out)
316 u_long which;
317 struct rt_metrics *in, *out;
318 {
319 #define metric(f, e) if (which & (f)) out->e = in->e;
320     metric(RTV_RPIPE, rmx_recvpipe);
321     metric(RTV_SPIPE, rmx_sendpipe);
322     metric(RTV_SSTHRESH, rmx_ssthresh);
323     metric(RTV_RTT, rmx_rtt);
324     metric(RTV_RTTVAR, rmx_rttvar);
325     metric(RTV_HOPCOUNT, rmx_hopcount);
326     metric(RTV_MTU, rmx_mtu);
327     metric(RTV_EXPIRE, rmx_expire);
328 #undef metric
329 }
```

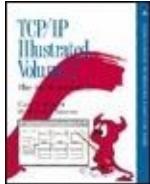
314-318

The which argument is always the rtm\_inits member of the routing message from the process. in points to the

`rt_metrics` structure from the process, and `out` points to the `rt_metrics` structure in the routing table entry that is being created or modified.

319-329

Each of the 8 bits in the bitmask is tested and if set, the corresponding metric is copied. Notice that when a new routing table entry is being created with the `RTM_ADD` command, `route_output` calls `rtrequest`, which sets the entire routing table entry to 0 ([Figure 19.9](#)). Hence, any metrics not specified by the process in the routing message default to 0.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.8 raw\_input Function

All routing messages destined for a process those that originate from within the kernel and those that originate from a process are given to `raw_input`, which selects the processes to receive the message. [Figure 18.11](#) summarizes the four functions that call `raw_input`.

When a routing socket is created, the family is always `PF_ROUTE` and the protocol, the third argument to `socket`, can be 0, which means the process wants to receive all routing messages, or a value such as `AF_INET`, which restricts the socket to messages containing addresses of that specific protocol family. A routing

control block is created for each routing socket (Section 20.3) and these two values are stored in the sp\_family and sp\_protocol members of the rcb\_proto structure.

[Figure 20.17](#) shows the raw\_input function.

**Figure 20.17. raw\_input function: pass routing messages to 0 or more processes.**

---

```
51 void  
52 raw_input(m0, proto, src, dst) raw_usrreq.c  
53 struct mbuf *m0;  
54 struct sockproto *proto;  
55 struct sockaddr *src, *dst;  
56 {  
57     struct rawcb *rp;  
58     struct mbuf *m = m0;  
59     int    sockets = 0;  
60     struct socket *last;
```

```

61     last = 0;
62     for (rp = rawcb.rcb_next; rp != &rawcb; rp = rp->rcb_next) {
63         if (rp->rcb_proto.sp_family != proto->sp_family)
64             continue;
65         if (rp->rcb_proto.sp_protocol &&
66             rp->rcb_proto.sp_protocol != proto->sp_protocol)
67             continue;
68         /*
69          * We assume the lower level routines have
70          * placed the address in a canonical format
71          * suitable for a structure comparison.
72          *
73          * Note that if the lengths are not the same
74          * the comparison will fail at the first byte.
75          */
76 #define equal(a1, a2) \
77     (bcmpl((caddr_t)(a1), (caddr_t)(a2), a1->sa_len) == 0)
78     if (rp->rcb_laddr && !equal(rp->rcb_laddr, dst))
79         continue;
80     if (rp->rcb_faddr && !equal(rp->rcb_faddr, src))
81         continue;
82     if (last) {
83         struct mbuf *n;
84         if (n = m_copy(m, 0, (int) M_COPYALL)) {
85             if (sbappendaddr(&last->so_rcv, src,
86                             n, (struct mbuf *) 0) == 0)
87                 /* should notify about lost packet */
88                 m_freem(n);
89             else {
90                 sorwakeup(last);
91                 sockets++;
92             }
93         }
94     }
95     last = rp->rcb_socket;
96 }
97 if (last) {
98     if (sbappendaddr(&last->so_rcv, src,
99                     m, (struct mbuf *) 0) == 0)
100        m_freem(m);
101    else {
102        sorwakeup(last);
103        sockets++;
104    }
105 } else
106     m_freem(m);
107 }

```

*raw\_usrreq.c*

## 51-61

In all four calls to `raw_input` that we've seen, the `proto`, `src`, and `dst` arguments are pointers to the three globals `route_proto`, `route_src`, and `route_dst`, which are declared and initialized as shown with [Figure 19.26](#).

## Compare address family and protocol

62-67

The for loop goes through every routing control block checking for a match. The family in the control block (normally PF\_ROUTE) must match the family in the sockproto structure or the control block is skipped. Next, if the protocol in the control block (the third argument to socket) is nonzero, it must match the family in the sockproto structure, or the message is skipped. Hence a process that creates a routing socket with a protocol of 0 receives all routing messages.

## Compare local and foreign addresses

68-81

These two tests compare the local address in the control block and the foreign address in the control block, if specified. Currently the process is unable to set the rcb\_laddr or rcb\_faddr members of the control block. Normally a process would

set the former with bind and the latter with connect, but that is not possible with routing sockets in Net/3. Instead, we'll see that route\_usrreq permanently connects the socket to the route\_src socket address structure, which is OK since that is always the src argument to this function.

## Append message to socket receive buffer

82-107

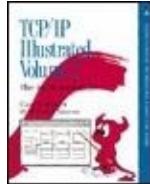
If last is nonnull, it points to the most recently seen socket structure that should receive this message. If this variable is nonnull, a copy of the message is appended to that socket's receive buffer by m\_copy and sbappendaddr, and any processes waiting on this receive buffer are awakened. Then last is set to point to this socket that just matched the previous tests. The use of last is to avoid calling m\_copy (an expensive operation) if only one process is to receive the message.

If  $N$  processes are to receive the message,

the first  $N - 1$  receive a copy and the final one receives the message itself.

The variable `sockets` that is incremented within this function is not used. Since it is incremented only when a message is passed to a process, if it is 0 at the end of the function it indicates that no process received the message (but the value isn't stored anywhere).





TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.9 route\_usrreq Function

route\_usrreq is the routing protocol's user-request function. It is called for a variety of operations. [Figure 20.18](#) shows the function.

**Figure 20.18. route\_usrreq function:  
process PRU\_xxx requests.**

---

```
64 int
65 route_usrreq(so, req, m, nam, control)
66 struct socket *so;
67 int      req;
68 struct mbuf *m, *nam, *control;
69 {
```

```

70     int      error = 0;
71     struct rawcb *rp = sotorawcb(so);
72     int      s;

73     if (req == PRU_ATTACH) {
74         MALLOC(rp, struct rawcb *, sizeof(*rp), M_PCB, M_WAITOK);
75         if (so->so_pcb = (caddr_t) rp)
76             bzero(so->so_pcb, sizeof(*rp));
77     }
78     if (req == PRU_DETACH && rp) {
79         int      af = rp->rcb_proto.sp_protocol;
80         if (af == AF_INET)
81             route_cb.ip_count--;
82         else if (af == AF_NS)
83             route_cb.ns_count--;
84         else if (af == AF_ISO)
85             route_cb.iso_count--;
86         route_cb.any_count--;
87     }
88     s = splnet();
89     error = raw_usrreq(so, req, m, nam, control);
90     rp = sotorawcb(so);
91     if (req == PRU_ATTACH && rp) {
92         int      af = rp->rcb_proto.sp_protocol;
93         if (error) {
94             free((caddr_t) rp, M_PCB);
95             splx(s);
96             return (error);
97         }
98         if (af == AF_INET)
99             route_cb.ip_count++;
100        else if (af == AF_NS)
101            route_cb.ns_count++;
102        else if (af == AF_ISO)
103            route_cb.iso_count++;
104        route_cb.any_count++;

105        rp->rcb_faddr = &route_src;
106        soisconnected(so);
107        so->so_options |= SO_USELOOPBACK;
108    }
109    splx(s);
110    return (error);
111 }

```

*rtsock.c*

## PRU\_ATTACH: allocate control block

64-77

The PRU\_ATTACH request is issued when the process calls socket. Memory is allocated for a routing control block. The pointer returned by MALLOC is stored in

the so\_pcb member of the socket structure, and if the memory was allocated, the rawcb structure is set to 0.

## **PRU\_DETACH: decrement counters**

78-87

The close system call issues the PRU\_DETACH request. If the socket structure points to a protocol control block, two of the counters in the route\_cb structure are decremented: one is the any\_count and one is based on the protocol.

## **Process request**

88-90

The function raw\_usrreq is called to process the PRU\_xxx request further.

## **Increment counters**

91-104

If the request is PRU\_ATTACH and the socket points to a routing control block, a check is made for an error from raw\_usrreq. Two of the counters in the route\_cb structure are then incremented: one is the any\_count and one is based on the protocol.

## Connect socket

105-106

The foreign address in the routing control block is set to route\_src. This permanently connects the new socket to receive routing messages from the PF\_ROUTE family.

## Enable SO\_USELOOPBACK by default

107-111

The SO\_USELOOPBACK socket option is enabled. This is a socket option that defaults to being enabled all others default to being disabled.

---

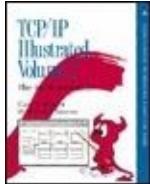
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.10 raw\_usrreq Function

raw\_usrreq performs most of the processing for the user request in the routing domain. It was called by route\_usrreq in the previous section. The reason the user-request processing is divided between these two functions is that other protocols (e.g., the OSI CLNP) call raw\_usrreq but not route\_usrreq. raw\_usrreq is not intended to be the pr\_usrreq function for a protocol. Instead it is a common subroutine called by the various pr\_usrreq functions.

Figure 20.19 shows the beginning and end of the raw\_usrreq function. The body of the switch is discussed in separate figures

following this figure.

## Figure 20.19. Body of raw\_usrreq function.

```
----- raw_usrreq.c
119 int
120 raw_usrreq(so, req, m, nam, control)
121 struct socket *so;
122 int     req;
123 struct mbuf *m, *nam, *control;
124 {
125     struct rawcb *rp = sotorawcb(so);
126     int     error = 0;
127     int     len;
128
129     if (req == PRU_CONTROL)
130         return (EOPNOTSUPP);
131     if (control && control->m_len) {
132         error = EOPNOTSUPP;
133         goto release;
134     }
135     if (rp == 0) {
136         error = EINVAL;
137         goto release;
138     }
139     switch (req) {
140
141         /* switch cases */
142
143     default:
144         panic("raw_usrreq");
145     }
146     release:
147     if (m != NULL)
148         m_free(m);
149     return (error);
150 }
----- raw_usrreq.c
```

## PRU\_CONTROL requests invalid

119-129

The PRU\_CONTROL request is from the ioctl system call and is not supported in

the routing domain.

## Control information invalid

130-133

If control information was passed by the process (using the sendmsg system call) an error is returned, since the routing domain doesn't use this optional information.

## Socket must have a control block

134-137

If the socket structure doesn't point to a routing control block, an error is returned. If a new socket is being created, it is the caller's responsibility (i.e., route\_usrreq) to allocate this control block and store the pointer in the so\_pcb member before calling this function.

262-269

The default for this switch catches two

requests that are not handled by case statements: PRU\_BIND and PRU\_CONNECT. The code for these two requests is present but commented out in Net/3. Therefore issuing the bind or connect system calls on a routing socket causes a kernel panic. This is a bug. Fortunately it requires a superuser process to create this type of socket.

We now discuss the individual case statements. [Figure 20.20](#) shows the processing for the PRU\_ATTACH and PRU\_DETACH requests.

**Figure 20.20. raw\_usrreq function:  
PRU\_ATTACH and PRU\_DETACH  
requests.**

```
139      /*
140       * Allocate a raw control block and fill in the
141       * necessary info to allow packets to be routed to
142       * the appropriate raw interface routine.
143       */
144     case PRU_ATTACH:
145       if ((so->so_state & SS_PRIV) == 0) {
146         error = EACCES;
147         break;
148       }
149       error = raw_attach(so, (int) nam);
150       break;

151      /*
152       * Destroy state just before socket deallocation.
153       * Flush data or not depending on the options.
154       */
155     case PRU_DETACH:
156       if (rp == 0) {
157         error = ENOTCONN;
158         break;
159       }
160       raw_detach(rp);
161       break;
```

raw\_usrreq.c

## 139-148

The PRU\_ATTACH request is a result of the socket system call. A routing socket must be created by a superuser process.

## 149-150

The function `raw_attach` ([Figure 20.24](#)) links the control block into the doubly linked list. The `nam` argument is the third argument to `socket` and gets stored in the control block.

## 151-159

The PRU\_DETACH is issued by the close system call. The test of a null `rp` pointer is

superfluous, since the test was already done before the switch statement.

160-161

raw\_detach ([Figure 20.25](#)) removes the control block from the doubly linked list.

[Figure 20.21](#) shows the processing of the PRU\_CONNECT2, PRU\_DISCONNECT, and PRU\_SHUTDOWN requests.

### **Figure 20.21. raw\_usrreq function: PRU\_CONNECT2, PRU\_DISCONNECT, and PRU\_SHUTDOWN requests.**

```
186     case PRU_CONNECT2:
187         error = EOPNOTSUPP;
188         goto release;
189
190     case PRU_DISCONNECT:
191         if (rp->rcb_faddr == 0) {
192             error = ENOTCONN;
193             break;
194         }
195         raw_disconnect(rp);
196         soisdisconnected(so);
197         break;
198
199         /*
200          * Mark the connection as being incapable of further input.
201          */
202     case PRU_SHUTDOWN:
203         socantsendmore(so);
204         break;
```

186-188

The PRU\_CONNECT2 request is from the socketpair system call and is not supported in the routing domain.

189-196

Since a routing socket is always connected ([Figure 20.18](#)), the PRU\_DISCONNECT request is issued by close before the PRU\_DETACH request. The socket must already be connected to a foreign address, which is always true for a routing socket. raw\_disconnect and soisdisconnected complete the processing.

197-202

The PRU\_SHUTDOWN request is from the shutdown system call when the argument specifies that no more writes will be performed on the socket. socantsendmore disables further writes.

The most common request for a routing socket, PRU\_SEND, and the PRU\_ABORT and PRU\_SENSE requests are shown in [Figure 20.22](#).

## Figure 20.22. raw\_usrreq function: PRU\_SEND, PRU\_ABORT, and PRU\_SENSE requests.

```
203      /*
204       * Ship a packet out.  The appropriate raw output
205       * routine handles any massaging necessary.
206       */
207     case PRU_SEND:
208       if (nam) {
209         if (rp->rcb_faddr) {
210           error = EISCONN;
211           break;
212         }
213         rp->rcb_faddr = mtod(nam, struct sockaddr *);
214       } else if (rp->rcb_faddr == 0) {
215         error = ENOTCONN;
216         break;
217       }
218       error = (*so->so_proto->pr_output) (m, so);
219       m = NULL;
220       if (nam)
221         rp->rcb_faddr = 0;
222       break;
223     case PRU_ABORT:
224       raw_disconnect(rp);
225       soffree(so);
226       soisdisconnected(so);
227       break;
228     case PRU_SENSE:
229     /*
230      * stat: don't bother with a blocksize.
231      */
232     return (0);
```

203-217

The PRU\_SEND request is issued by sosend when the process writes to the socket. If a nam argument is specified, that is, the process specified a destination address using either sendto or sendmsg, an error is returned because route\_usrreq always sets rcb\_faddr for a routing socket.

218-222

The message in the mbuf chain pointed to by m is passed to the protocol's pr\_output function, which is route\_output.

223-227

If a PRU\_ABORT request is issued, the control block is disconnected, the socket is released, and the socket is disconnected.

228-232

The PRU\_SENSE request is issued by the fstat system call. The function returns OK.

[Figure 20.23](#) shows the remaining PRU\_xxx requests.

**Figure 20.23. raw\_usrreq function: final part.**

```
233     /*  
234      * Not supported.  
235      */  
236     case PRU_RCVOOB:  
237     case PRU_RECVD:  
238         return (EOPNOTSUPP);  
  
239     case PRU_LISTEN:  
240     case PRU_ACCEPT:  
241     case PRU_SENDOOB:  
242         error = EOPNOTSUPP;  
243         break;  
  
244     case PRU_SOCKADDR:  
245         if (rp->rcb_laddr == 0) {  
246             error = EINVAL;  
247             break;  
248         }  
249         len = rp->rcb_laddr->sa_len;  
250         bcopy((caddr_t) rp->rcb_laddr, mtod(nam, caddr_t), (unsigned) len);  
251         nam->m_len = len;  
252         break;  
  
253     case PRU_PEERADDR:  
254         if (rp->rcb_faddr == 0) {  
255             error = ENOTCONN;  
256             break;  
257         }  
258         len = rp->rcb_faddr->sa_len;  
259         bcopy((caddr_t) rp->rcb_faddr, mtod(nam, caddr_t), (unsigned) len);  
260         nam->m_len = len;  
261         break;
```

233-243

These five requests are not supported.

244-261

The PRU\_SOCKADDR and PRU\_PEERADDR requests are from the getsockname and getpeername system calls respectively. The former always returns an error, since the bind system call, which sets the local address, is not supported in the routing domain. The latter always returns the contents of the socket address structure

`route_src`, which was set by `route_usrreq` as the foreign address.

---

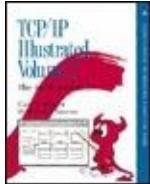
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.11 raw\_attach, raw\_detach, and raw\_disconnect Functions

The raw\_attach function, shown in [Figure 20.24](#), was called by raw\_input to finish processing the PRU\_ATTACH request.

**Figure 20.24. raw\_attach function.**

```
49 int  
50 raw_attach(so, proto)  
51 struct socket *so;  
52 int proto;  
53 {  
54     struct rawcb *rp = sotorawcb(so);  
55     int error;  
56     /*  
57      * It is assumed that raw_attach is called  
58      * after space has been allocated for the  
59      * rawcb.  
60      */  
61     if (rp == 0)  
62         return (ENOBUFS);  
63     if (error = soreserve(so, raw_sendspace, raw_recvspace))  
64         return (error);  
65     rp->rcb_socket = so;  
66     rp->rcb_proto.sp_family = so->so_proto->pr_domain->dom_family;  
67     rp->rcb_proto.sp_protocol = proto;  
68     insque(rp, &rawcb);  
69     return (0);  
70 }
```

raw\_cb.c

## 49-64

The caller must have already allocated the raw protocol control block. soreserve sets the high-water marks for the send and receive buffers to 8192. This should be more than adequate for the routing messages.

## 65-67

A pointer to the socket structure is stored in the protocol control block along with the dom\_family (which is PF\_ROUTE from [Figure 20.1](#) for the routing domain) and the proto argument (which is the third argument to socket).

68-70

insque adds the control block to the front of the doubly linked list headed by the global rawcb.

The raw\_detach function, shown in [Figure 20.25](#), was called by raw\_input to finish processing the PRU\_DETACH request.

### [Figure 20.25. raw\\_detach function.](#)

```
75 void  
76 raw_detach(rp)  
77 struct rawcb *rp;  
78 {  
79     struct socket *so = rp->rcb_socket;  
80     so->so_pcb = 0;  
81     sofree(so);  
82     remque(rp);  
83     free((caddr_t) (rp), M_PCB);  
84 }
```

75-84

The so\_pcb pointer in the socket structure is set to null and the socket is released. The control block is removed from the doubly linked list by remque and the memory used for the control block is released by free.

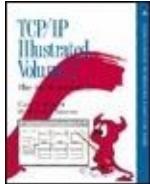
The raw\_disconnect function, shown in [Figure 20.26](#), was called by raw\_input to process the PRU\_DISCONNECT and PRU\_ABORT requests.

## Figure 20.26. raw\_disconnect function.

```
88 void  
89 raw_disconnect(rp)  
90 struct rawcb *rp;  
91 {  
92     if (rp->rcb_socket->so_state & SS_NOFDREF)  
93         raw_detach(rp);  
94 }
```

88-94

If the socket does not reference a descriptor, raw\_detach releases the socket and control block.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 20. Routing Sockets

### 20.12 Summary

A routing socket is a raw socket in the PF\_ROUTE domain. Routing sockets can be created only by a superuser process. If a nonprivileged process wants to read the routing information contained in the kernel, the sysctl system call supported by the routing domain can be used (we described this in the previous chapter).

This chapter was our first encounter with the protocol control blocks (PCBs) that are normally associated with each socket. In the routing domain a special rawcb contains information about the routing socket: the local and foreign addresses, the address family, and the protocol. We'll

see in [Chapter 22](#) that the larger Internet protocol control block (inpcb) is used with UDP, TCP, and raw IP sockets. The concepts are the same, however: the socket structure is used by the socket layer, and the PCB, a rawcb or an inpcb, is used by the protocol layer. The socket structure points to the PCB and vice versa.

The `route_output` function handles the five routing requests that can be issued by a process. `raw_input` delivers a routing message to one or more routing sockets, depending on the protocol and address family. The various `PRU_xxx` requests for a routing socket are handled by `raw_usrreq` and `route_usrreq`. In later chapters we'll encounter additional `xxx_usrreq` functions, one per protocol (UDP, TCP, and raw IP), each consisting of a switch statement to handle each request.

## Exercises

List two ways a process can receive the return value from `route_output`

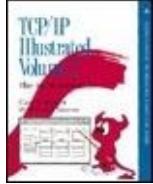
**20.1** when the process writes a message to a routing socket. Which method is more reliable?

What happens when a process specifies a nonzero *protocol*

**20.2** argument to the socket system call, since the pr\_protocol member of the routesw structure is 0?

Routes in the routing table (other than ARP entries) never time out.  
**20.3** Implement a timeout on routes.





[\*\*TCP/IP Illustrated, Volume 2: The Implementation\*\*](#)  
By Gary R. Wright,  
W. Richard Stevens  
[\*\*Table of Contents\*\*](#)

# Chapter 21. ARP: Address Resolution Protocol

[Section 21.1. Introduction](#)

[Section 21.2. ARP and the Routing Table](#)

[Section 21.3. Code Introduction](#)

[Section 21.4. ARP Structures](#)

[Section 21.5. arpwhohas Function](#)

[Section 21.6. arprequest Function](#)

[Section 21.7. arpintr Function](#)

[Section 21.8. in\\_arpinput Function](#)

[Section 21.9. ARP Timer Functions](#)

[Section 21.10. arpresolve Function](#)

[Section 21.11. arplookup Function](#)

[Section 21.12. Proxy ARP](#)

[Section 21.13. arp\\_rtrequest Function](#)

[Section 21.14. ARP and Multicasting](#)

[Section 21.15. Summary](#)

---

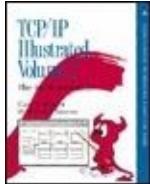
**Team-Fly**



[Previous](#)

[Next](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

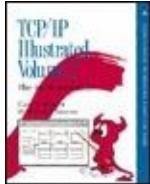
### 21.1 Introduction

ARP, the Address Resolution Protocol, handles the translation of 32-bit IP addresses into the corresponding hardware address. For an Ethernet, the hardware addresses are 48-bit Ethernet addresses. In this chapter we only consider mapping IP addresses into 48-bit Ethernet addresses, although ARP is more general and can work with other types of data links. ARP is specified in RFC 826 [Plummer 1982].

When a host has an IP datagram to send to another host on a locally attached Ethernet, the local host first looks up the

destination host in the *ARP cache*, a table that maps a 32-bit IP address into its corresponding 48-bit Ethernet address. If the entry is found for the destination, the corresponding Ethernet address is copied into the Ethernet header and the datagram is added to the appropriate interface's output queue. If the entry is not found, the ARP functions hold onto the IP datagram, broadcast an ARP request asking the destination host for its Ethernet address, and, when a reply is received, send the datagram to its destination.

This simple overview handles the common case, but there are many details that we describe in this chapter as we examine the Net/3 implementation of ARP. Chapter 4 of Volume 1 contains additional ARP examples.



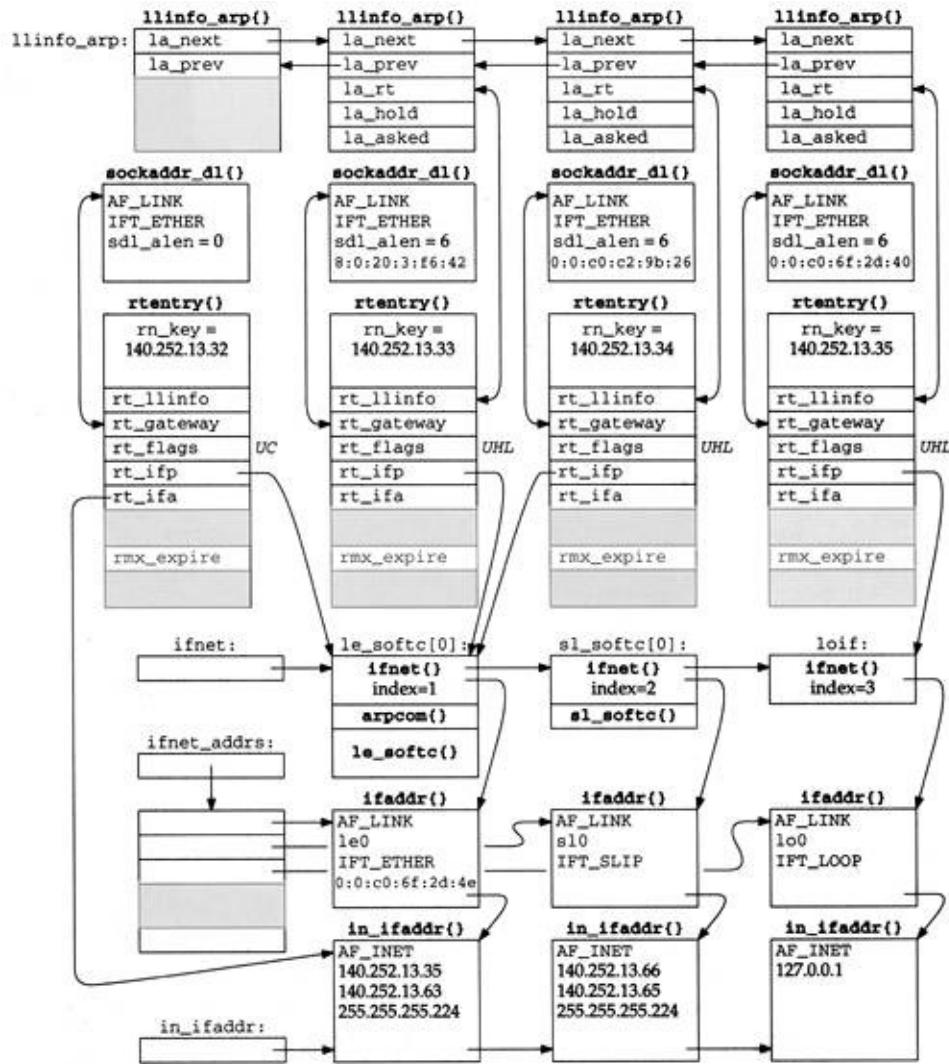
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

## 21.2 ARP and the Routing Table

The Net/3 implementation of ARP is tied to the routing table, which is why we postponed discussing ARP until we had described the structure of the Net/3 routing tables. [Figure 21.1](#) shows an example that we use in this chapter when describing ARP.

**Figure 21.1. Relationship of ARP to routing table and interface structures.**



The entire figure corresponds to the example network used throughout the text ([Figure 1.17](#)). It shows the ARP entries on the system bsdi. The ifnet, ifaddr, and in\_ifaddr structures are simplified from [Figures 3.32](#) and [6.5](#). We have removed some of the details from these three structures, which were covered in [Chapters 3](#) and [6](#).

For example, we don't show the two `sockaddr_dl` structures that appear after each `ifaddr` structure instead we summarize the information contained in these two structures. Similarly, we summarize the information contained in the three `in_ifaddr` structures.

We briefly summarize some relevant points from this figure, the details of which we cover as we proceed through the chapter.

- 1. A doubly linked list of `llinfo_arp` structures contains a minimal amount of information for each hardware address known by ARP. The global `llinfo_arp` is the head of this list. Not shown in this figure is that the `la_prev` pointer of the first entry points to the last entry, and the `la_next` pointer of the last entry points to the first entry. This linked list is processed by the ARP timer function every 5 minutes.**
  - For each IP address with a known hardware address, a routing table entry exists (an `rtentry` structure). The `llinfo_arp`

structure points to the corresponding `rtentry` structure, and vice versa, using the `la_rt` and `rt_llinfo` pointers. The three routing table entries in this figure with an associated `llinfo_arp` structure are for the hosts sun (140.252.13.33), svr4 (140.252.13.34), and bsdi itself (140.252.13.35). These three are also shown in [Figure 18.2](#).

- We show a fourth routing table entry on the left, without an `llinfo_arp` structure, which is the entry for the network route to the local Ethernet (140.252.13.32). We show its `rt_flags` with the C bit on, since this entry is cloned to form the other three routing table entries. This entry is created by the call to `rtinit` when the IP address is assigned to the interface by `in_ifinit` ([Figure 6.19](#)). The other three entries are host entries (the H flag) and are generated by ARP (the L flag) when a datagram is sent to that IP address.
- The `rt_gateway` member of the `rtentry` structure points to a `sockaddr_dl` structure. This data-link socket address structure contains the hardware address if

the `sdl_alen` member equals 6.

- The `rt_ifp` member of the routing table entry points to the `ifnet` structure of the outgoing interface. Notice that the two routing table entries in the middle, for other hosts on the local Ethernet, both point to `le_softc[0]`, but the routing table entry on the right, for the host `bsdi` itself, points to the loopback structure. Since `rt_ifp.if_output` ([Figure 8.25](#)) points to the output routine, packets sent to the local IP address are routed to the loopback interface.
- Each routing table entry also points to the corresponding `in_ifaddr` structure. (Actually the `rt_ifa` member points to an `ifaddr` structure, but recall from [Figure 6.8](#) that the first member of an `in_ifaddr` structure is an `ifaddr` structure.) We show only one of these pointers in the figure, although all four point to the same structure. Remember that a single interface, say `le0`, can have multiple IP addresses, each with its own `in_ifaddr` structure, which is why the `rt_ifa` pointer is required in addition to the `rt_ifp` pointer.

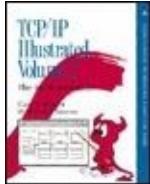
- The la\_hold member is a pointer to an mbuf chain. An ARP request is broadcast because a datagram is sent to that IP address. While the kernel awaits the ARP reply it holds onto the mbuf chain for the datagram by storing its address in la\_hold. When the ARP reply is received, the mbuf chain pointed to by la\_hold is sent.
- Finally, we show the variable rmx\_expire, which is in the rt\_metrics structure within the routing table entry. This value is the timer associated with each ARP entry. Some time after an ARP entry has been created (normally 20 minutes) the ARP entry is deleted.

Even though major routing table changes took place with 4.3BSD Reno, the ARP cache was left alone with 4.3BSD Reno and Net/2. 4.4BSD, however, removed the stand-alone ARP cache and moved the ARP information into the routing table.

The ARP table in Net/2 was an array of structures composed of the following members: an IP address, an Ethernet

address, a timer, flags, and a pointer to an mbuf (similar to the la\_hold member in [Figure 21.1](#)). We see with Net/3 that the same information is now spread throughout multiple structures, all of which are linked.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.3 Code Introduction

There are nine ARP functions in a single C file and definitions in two headers, as shown in [Figure 21.2](#).

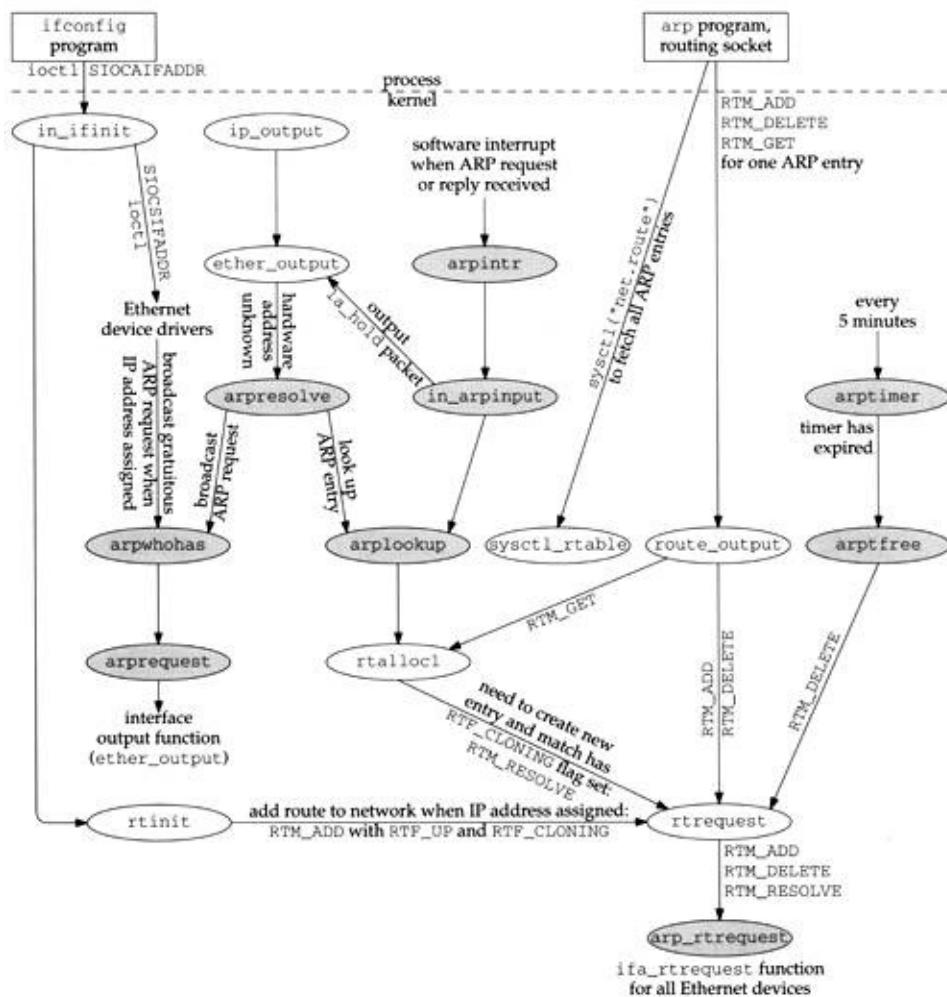
**Figure 21.2. Files discussed in this chapter.**

File	Description
<code>net/if_arp.h</code>	<code>arp_hdr</code> structure definition
<code>netinet/if_ether.h</code>	various structure and constant definitions
<code>netinet/if_ether.c</code>	ARP functions

[Figure 21.3](#) shows the relationship of the ARP functions to other kernel functions. In this figure we also show the relationship

between the ARP functions and some of the routing functions from [Chapter 19](#). We describe all these relationships as we proceed through the chapter.

## Figure 21.3. Relationship of ARP functions to rest of kernel.



## Global Variables

Ten global variables are introduced in this chapter, which are shown in [Figure 21.4](#).

**Figure 21.4. Global variables introduced in this chapter.**

Variable	Datatype	Description
llinfo_arp	struct llinfo_arp	head of llinfo_arp doubly linked list (Figure 21.1)
arpintrq	struct ifqueue	ARP input queue from Ethernet device drivers (Figure 4.9)
arpt_prune	int	#seconds between checking ARP list ( $5 \times 60$ )
arpt_keep	int	#seconds ARP entry valid once resolved ( $20 \times 60$ )
arpt_down	int	#seconds between ARP flooding algorithm (20)
arp_inuse	int	#ARP entries currently in use
arp_allocated	int	#ARP entries ever allocated
arp_maxtries	int	max #tries for an IP address before pausing (5)
arpinit_done	int	initialization-performed flag
useloopback	int	use loopback for local host (default true)

## Statistics

The only statistics maintained by ARP are the two globals arp\_inuse and arp\_allocated, from [Figure 21.4](#). The former counts the number of ARP entries currently in use and the latter counts the total number of ARP entries allocated since the system was initialized. Neither counter is output by the netstat program, but they can be examined with a debugger.

The entire ARP cache can be listed using the arp -a command, which uses the sysctl system call with the arguments shown in [Figure 19.36](#). [Figure 21.5](#) shows the output from this command, for the entries shown in [Figure 18.2](#).

## **Figure 21.5. arp -a output corresponding to Figure 18.2.**

```
bsdi $ arp -a
sun.tuc.noao.edu (140.252.13.33) at 8:0:20:3:f6:42
svr4.tuc.noao.edu (140.252.13.34) at 0:0:c0:c2:9b:26
bsdi.tuc.noao.edu (140.252.13.35) at 0:0:c0:6f:2d:40 permanent
ALL-SYSTEMS.MCAST.NET (224.0.0.1) at (incomplete)
```

Since the multicast group 224.0.0.1 has the L flag set in [Figure 18.2](#), and since the arp program looks for entries with the RTF\_LLINFO flag set, the multicast groups are output by the program. Later in this chapter we'll see why this entry is marked as "incomplete" and why the entry above it is "permanent."

## **SNMP Variables**

As described in Section 25.8 of Volume 1,

the original SNMP MIB defined an address translation group that was the system's ARP cache. MIB-II deprecated this group and instead each network protocol group (i.e., IP) contains its own address translation tables. Notice that the change in Net/2 to Net/3 from a stand-alone ARP table to an integration of the ARP information within the IP routing table parallels this SNMP change.

[Figure 21.6](#) shows the IP address translation table from MIB-II, named `ipNetToMediaTable`. The values returned by SNMP for this table are taken from the routing table entry and its corresponding `ifnet` structure.

## Figure 21.6. IP address translation table: `ipNetToMediaTable`.

IP address translation table, index = < <code>ipNetToMediaIfIndex</code> >.< <code>ipNetToMediaNetAddress</code> >		
Name	Member	Description
<code>ipNetToMediaIfIndex</code>	<code>if_index</code>	corresponding interface: <code>ifIndex</code>
<code>ipNetToMediaPhysAddress</code>	<code>rt_gateway</code>	physical address
<code>ipNetToMediaNetAddress</code>	<code>rt_key</code>	IP address
<code>ipNetToMediaType</code>	<code>rt_flags</code>	type of mapping: 1 = other, 2 = invalidated, 3 = dynamic, 4 = static (see text)

If the routing table entry has an expiration

time of 0 it is considered permanent and hence "static." Otherwise the entry is considered "dynamic."

---

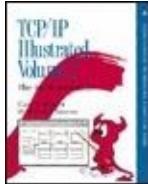
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



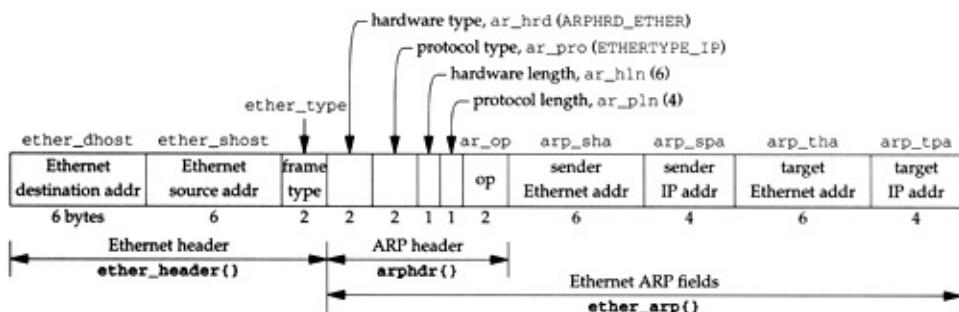
TCP/IP Illustrated, Volume 2: The Implementation By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.4 ARP Structures

Figure 21.7 shows the format of an ARP packet when transmitted on an Ethernet.

**Figure 21.7. Format of an ARP request or reply when used on an Ethernet.**



The `ether_header` structure (Figure 4.10) defines the 14-byte Ethernet header; the

arphdr structure defines the next five fields, which are common to ARP requests and ARP replies on any type of media; and the ether\_arp structure combines the arphdr structure with the sender and target addresses when ARP is used on an Ethernet.

[Figure 21.8](#) shows the definition of the arphdr structure. [Figure 21.7](#) shows the values of the first four fields in this structure when ARP is mapping IP addresses to Ethernet addresses.

### Figure 21.8. arphdr structure: common ARP request/reply header.

```
45 struct arphdr {  
46     u_short ar_hrd;          /* format of hardware address */  
47     u_short ar_pro;          /* format of protocol address */  
48     u_char  ar_hln;          /* length of hardware address */  
49     u_char  ar_pln;          /* length of protocol address */  
50     u_short ar_op;          /* ARP/RARP operation, Figure 21.15 */  
51 };
```

[Figure 21.9](#) shows the combination of the arphdr structure with the fields used with IP addresses and Ethernet addresses, forming the ether\_arp structure. Notice that ARP uses the terms *hardware* to

describe the 48-bit Ethernet address, and *protocol* to describe the 32-bit IP address.

## Figure 21.9. ether\_arp structure.

```
79 struct ether_arp {  
80     struct arphdr ea_hdr;      /* fixed-size header */  
81     u_char    arp_sha[6];      /* sender hardware address */  
82     u_char    arp_spa[4];      /* sender protocol address */  
83     u_char    arp_tha[6];      /* target hardware address */  
84     u_char    arp_tpa[4];      /* target protocol address */  
85 };  
  
86 #define arp_hrd ea_hdr.ar_hrd  
87 #define arp_pro ea_hdr.ar_pro  
88 #define arp_hln ea_hdr.ar_hln  
89 #define arp_pln ea_hdr.ar_pln  
90 #define arp_op  ea_hdr.ar_op  
if_ether.h  
if_ether.h
```

One `llinfo_arp` structure, shown in [Figure 21.10](#), exists for each ARP entry.

Additionally, one of these structures is allocated as a global of the same name and used as the head of the linked list of all these structures. We often refer to this list as the *ARP cache*, since it is the only data structure in [Figure 21.1](#) that has a one-to-one correspondence with the ARP entries.

## Figure 21.10. llinfo\_arp structure.

```
103 struct llinfo_arp {  
104     struct llinfo_arp *la_next;  
105     struct llinfo_arp *la_prev;  
106     struct rtentry *la_rt;  
107     struct mbuf *la_hold;      /* last packet until resolved/timeout */  
108     long    la_asked;        /* #times we've queried for this addr */  
109 };  
110 #define la_timer la_rt->rt_rmx.rmx_expire /* deletion time in seconds */  
-----if_ether.h-----
```

With Net/2 and earlier systems it was easy to identify the structure called the *ARP cache*, since a single structure contained everything for each ARP entry. Since Net/3 stores the ARP information among multiple structures, no single structure can be called the *ARP cache*. Nevertheless, having the concept of an ARP cache, which is the collection of information describing a single ARP entry, simplifies the discussion.

## 104-106

The first two entries form the doubly linked list, which is updated by the insque and remque functions. la\_rt points to the associated routing table entry, and the rt\_llinfo member of the routing table entry points to this structure.

## 107

When ARP receives an IP datagram to send to another host but the destination's hardware address is not in the ARP cache, an ARP request must be sent and the ARP reply received before the datagram can be sent. While waiting for the reply the mbuf pointer to the datagram is saved in `la_hold`. When the ARP reply is received, the packet pointed to by `la_hold` (if any) is sent.

## 108-109

`la_asked` counts how many consecutive times an ARP request has been sent to this IP address without receiving a reply. We'll see in [Figure 21.24](#) that when this counter reaches a limit, that host is considered down and another ARP request won't be sent for a while.

## 110

This definition uses the `rmx_expire` member of the `rt_metrics` structure in the routing table entry as the ARP timer. When the value is 0, the ARP entry is considered permanent. When nonzero, the value is

the number of seconds since the Unix Epoch when the entry expires.

---

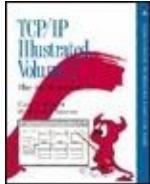
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.5 arpwhohas Function

The arpwhohas function is normally called by arpresolve to broadcast an ARP request. It is also called by each Ethernet device driver to issue a *gratuitous ARP* request when the IP address is assigned to the interface (the SIOCSIFADDR ioctl in [Figure 6.28](#)). Section 4.7 of Volume 1 describes gratuitous ARP it detects if another host on the Ethernet is using the same IP address and also allows other hosts with ARP entries for this host to update their ARP entry if this host has changed its Ethernet address. arpwhohas simply calls arprequest, shown in the next section, with the correct arguments.

## Figure 21.11. arpwhohas function: broadcast an ARP request.

```
196 void  
197 arpwhohas(ac, addr)  
198 struct arpcom *ac;  
199 struct in_addr *addr;  
200 {  
201     arprequest(ac, &ac->ac_ipaddr.s_addr, &addr->s_addr, ac->ac_enaddr);  
202 }
```

196-202

The arpcom structure ([Figure 3.26](#)) is common to all Ethernet devices and is part of the le\_softc structure, for example ([Figure 3.20](#)). The ac\_ipaddr member is a copy of the interface's IP address, which is set by the driver when the SIOCSIFADDR ioctl is executed ([Figure 6.28](#)). ac\_enaddr is the Ethernet address of the device.

The second argument to this function, addr, is the IP address for which the ARP request is being issued: the target IP address. In the case of a gratuitous ARP request, addr equals ac\_ipaddr, so the second and third arguments to arprequest are the same, which means the sender IP address will equal the target IP address in the gratuitous ARP request.

---

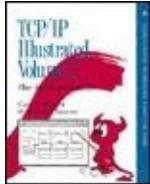
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.6 arprequest Function

The arprequest function is called by arpwho has to broadcast an ARP request. It builds an ARP request packet and passes it to the interface's output function.

Before looking at the source code, let's examine the data structures built by the function. To send the ARP request the interface output function for the Ethernet device (`ether_output`) is called. One argument to `ether_output` is an mbuf containing the data to send: everything that follows the Ethernet type field in [Figure 21.7](#). Another argument is a socket address structure containing the

destination address. Normally this destination address is an IP address (e.g., when `ip_output` calls `ether_output` in [Figure 21.3](#)). For the special case of an ARP request, the `sa_family` member of the socket address structure is set to `AF_UNSPEC`, which tells `ether_output` that it contains a filled-in Ethernet header, including the destination Ethernet address. This prevents `ether_output` from calling `arpresolve`, which would cause an infinite loop. We don't show this loop in [Figure 21.3](#), but the "interface output function" below `arprequest` is `ether_output`. If `ether_output` were to call `arpresolve` again, the infinite loop would occur.

[Figure 21.12](#) shows the mbuf and the socket address structure built by this function. We also show the two pointers `eh` and `ea`, which are used in the function.

**Figure 21.12. sockaddr and mbuf built by arprequest.**

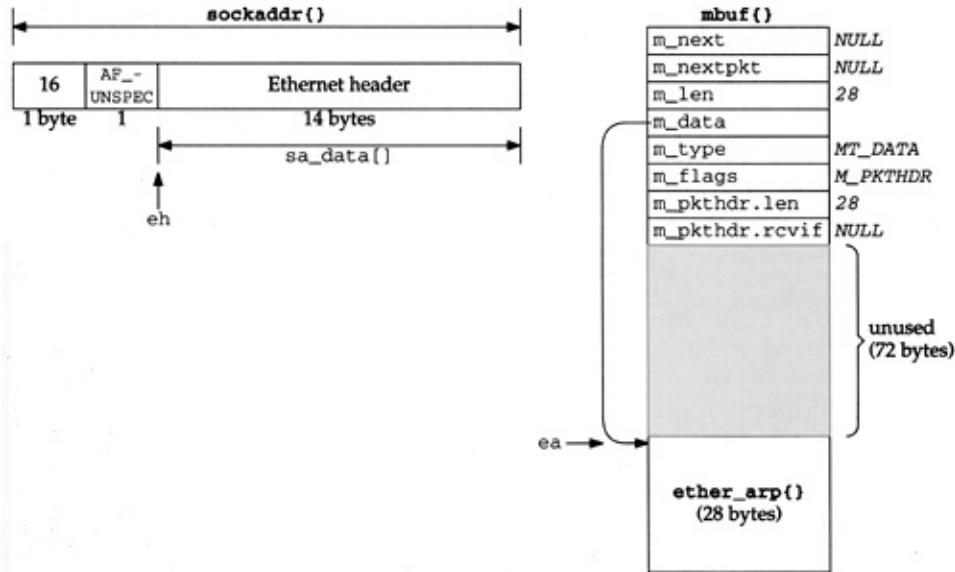


Figure 21.13 shows the arprequest function.

**Figure 21.13. arprequest function: build an ARP request packet and send it.**

```

209 static void
210 arprequest(ac, sip, tip, enaddr)
211 struct arpcom *ac;
212 u_long *sip, *tip;
213 u_char *enaddr;
214 {
215     struct mbuf *m;
216     struct ether_header *eh;
217     struct ether_arp *ea;
218     struct sockaddr sa;
219
220     if ((m = m_gethdr(M_DONTWAIT, MT_DATA)) == NULL)
221         return;
222     m->m_len = sizeof(*ea);
223     m->m_pkthdr.len = sizeof(*ea);
224     MH_ALIGN(m, sizeof(*ea));
225
226     ea = mtod(m, struct ether_arp *);
227     eh = (struct ether_header *) sa.sa_data;
228     bzero((caddr_t) ea, sizeof(*ea));
229
230     bcopy((caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
231           sizeof(eh->ether_dhost));
232     eh->ether_type = ETHERTYPE_ARP; /* if_output() will swap */
233
234     ea->arp_hrd = htons(ARPHRD_ETHER);
235     ea->arp_pro = htons(ETHERTYPE_IP);
236     ea->arp_hln = sizeof(ea->arp_sha); /* hardware address length */
237     ea->arp_pln = sizeof(ea->arp_spa); /* protocol address length */
238     ea->arp_op = htons(ARPOP_REQUEST);
239
240     bcopy((caddr_t) enaddr, (caddr_t) ea->arp_sha, sizeof(ea->arp_sha));
241     bcopy((caddr_t) sip, (caddr_t) ea->arp_spa, sizeof(ea->arp_spa));
242     bcopy((caddr_t) tip, (caddr_t) ea->arp_tpa, sizeof(ea->arp_tpa));
243
244     sa.sa_family = AF_UNSPEC;
245     sa.sa_len = sizeof(sa);
246
247     (*ac->ac_if.if_output) (&ac->ac_if, m, &sa, (struct rtentry *) 0);
248 }

```

if\_ether.c

## Allocate and initialize mbuf

209-223

A packet header mbuf is allocated and the two length fields are set. MH\_ALIGN allows room for a 28-byte ether\_arp structure at the end of the mbuf, and sets the m\_data pointer accordingly. The reason for moving this structure to the end of the mbuf is to

allow ether\_output to prepend the 14-byte Ethernet header in the same mbuf.

## Initialize pointers

224-226

The two pointers ea and eh are set and the ether\_arp structure is set to 0. The only purpose of the call to bzero is to set the target hardware address to 0, because the other eight fields in this structure are explicitly set to their respective value.

## Fill in Ethernet header

227-229

The destination Ethernet address is set to the Ethernet broadcast address and the Ethernet type field is set to ETHERTYPE\_ARP. Note the comment that this 2-byte field will be converted from host byte order to network byte order by the interface output function. This function also fills in the Ethernet source address field. [Figure 21.14](#) shows the different

values for the Ethernet type field.

**Figure 21.14. Ethernet type fields.**

Constant	Value	Description
<code>ETHERTYPE_IP</code>	0x0800	IP frames
<code>ETHERTYPE_ARP</code>	0x0806	ARP frames
<code>ETHERTYPE_REVARP</code>	0x8035	reverse ARP (RARP) frames
<code>ETHERTYPE_IPTAILERS</code>	0x1000	trailer encapsulation (deprecated)

RARP maps an Ethernet address to an IP address and is used when a diskless system bootstraps. RARP is normally not part of the kernel's implementation of TCP/IP, so it is not covered in this text. Chapter 5 of Volume 1 describes RARP.

## Fill in ARP fields

230-237

All fields in the `ether_arp` structure are filled in, except the target hardware address, which is what the ARP request is looking for. The constant `ARPHRD_ETHER`, which has a value of 1, specifies the format of the hardware addresses as 6-byte Ethernet addresses. To identify the

protocol addresses as 4-byte IP addresses, arp\_pro is set to the Ethernet type field for IP from [Figure 21.14](#). [Figure 21.15](#) shows the various ARP operation codes. We encounter the first two in this chapter. The last two are used with RARP.

**Figure 21.15. ARP operation codes.**

Constant	Value	Description
ARPOP_REQUEST	1	ARP request to resolve protocol address
ARPOP_REPLY	2	reply to ARP request
ARPOP_REVREQUEST	3	RARP request to resolve hardware address
ARPOP_REVREPLY	4	reply to RARP request

**Fill in sockaddr and call interface output function**

238-241

The sa\_family member of the socket address structure is set to AF\_UNSPEC and the sa\_len member is set to 16. The interface output function is called, which we said is ether\_output.

---

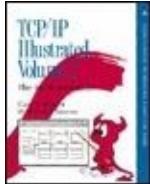
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.7 arpintr Function

In [Figure 4.13](#) we saw that when `ether_input` receives an Ethernet frame with a type field of `ETHERTYPE_ARP`, it schedules a software interrupt of priority `NETISR_ARP` and appends the frame to ARP's input queue: `arpintrq`. When the kernel processes the software interrupt, the function `arpintr`, shown in [Figure 21.16](#), is called.

**Figure 21.16. arpintr function: process Ethernet frames containing ARP requests or replies.**

```
319 void
320 arpintr()
321 {
322     struct mbuf *m;
323     struct arphdr *ar;
324     int     s;

325     while (arpintrq.ifq_head) {
326         s = splimp();
327         IF_DEQUEUE(&arpintrq, m);
328         splx(s);
329         if (m == 0 || (m->m_flags & M_PKTHDR) == 0)
330             panic("arpintr");

331         if (m->m_len >= sizeof(struct arphdr) &&
332             (ar = mtod(m, struct arphdr *)) &&
333             ntohs(ar->ar_hrd) == ARPHRD_ETHER &&
334             m->m_len >= sizeof(struct arphdr) + 2*ar->ar_hln + 2*ar->ar_pln)

335             switch (ntohs(ar->ar_pro)) {
336                 case ETHERTYPE_IP:
337                 case ETHERTYPE_IPTRAILERS:
338                     in_arpinput(m);
339                     continue;
340             }

341         m_freem(m);
342     }
343 }
```

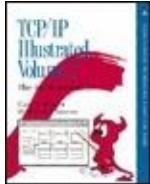
if\_ether.c

## 319-343

The while loop processes one frame at a time, as long as there are frames on the queue. The frame is processed if the hardware type specifies Ethernet addresses, and if the size of the frame is greater than or equal to the size of an arphdr structure plus the sizes of two hardware addresses and two protocol addresses. If the type of protocol addresses is either ETHERTYPE\_IP or ETHERTYPE\_IPTRAILERS, the in\_arpinput function, shown in the next section, is called. Otherwise the frame is discarded.

Notice the order of the tests within the if statement. The length is checked twice. First, if the length is at least the size of an arphdr structure, then the fields in that structure can be examined. The length is checked again, using the two length fields in the arphdr structure.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.8 in\_arpinput Function

This function is called by arpintr to process each received ARP request or ARP reply. While ARP is conceptually simple, numerous rules add complexity to the implementation. The following two scenarios are typical:

- 1. If a request is received for one of the host's IP addresses, a reply is sent. This is the normal case of some other host on the Ethernet wanting to send this host a packet. Also, since we're about to receive a packet from that other host, and we'll probably send a reply, an ARP**

**entry is created for that host (if one doesn't already exist) because we have its IP address and hardware address. This optimization avoids another ARP exchange when the packet is received from the other host.**

- If a reply is received in response to a request sent by this host, the corresponding ARP entry is now complete (the hardware address is known). The other host's hardware address is stored in the sockaddr\_dl structure and any queued packet for that host can now be sent. Again, this is the normal case.

ARP requests are normally broadcast so each host sees *all* ARP requests on the Ethernet, even those requests for which it is not the target. Recall from arprequest that when a request is sent, it contains the *sender's* IP address and hardware address. This allows the following tests also to occur.

### **3. If some other host sends a request or reply with a sender IP address**

**that equals this host's IP address, one of the two hosts is misconfigured. Net/3 detects this error and logs a message for the administrator. (We say "request or reply" here because `in_arpinput` doesn't examine the operation type. But ARP replies are normally unicast, in which case only the target host of the reply receives the reply.)**

- If this host receives a request or reply from some other host for which an ARP entry already exists, and if the other host's hardware address has changed, the hardware address in the ARP entry is updated accordingly. This can happen if the other host is shut down and then rebooted with a different Ethernet interface (hence a different hardware address) before its ARP entry times out. The use of this technique, along with the other host sending a gratuitous ARP request when it reboots, prevents this host from being unable to communicate with the other host after the reboot because of an ARP entry that is no longer valid.

- This host can be configured as a *proxy ARP server*. This means it responds to ARP requests for some other host, supplying the other host's hardware address in the reply. The host whose hardware address is supplied in the proxy ARP reply must be one that is able to forward IP datagrams to the host that is the target of the ARP request. Section 4.6 of Volume 1 discusses proxy ARP.

A Net/3 system can be configured as a proxy ARP server. These ARP entries are added with the arp command, specifying the IP address, hardware address, and the keyword pub. We'll see the support for this in [Figure 21.20](#) and we describe it in [Section 21.12](#).

We examine `in_arpinput` in four parts. [Figure 21.17](#) shows the first part.

**Figure 21.17. `in_arpinput` function: look for matching interface.**

```

358 static void
359 in_arpinput(m)
360 struct mbuf *m;
361 {
362     struct ether_arp *ea;
363     struct arpcom *ac = (struct arpcom *) m->m_pkthdr.rcvif;
364     struct ether_header *eh;
365     struct linfo_arp *la = 0;
366     struct rtentry *rt;
367     struct in_ifaddr *ia, *maybe_ia = 0;
368     struct sockaddr_dl *sdl;
369     struct sockaddr sa;
370     struct in_addr isaddr, itaddr, myaddr;
371     int op;

372     ea = mtod(m, struct ether_arp *);
373     op = ntohs(ea->arp_op);
374     bcopy((caddr_t) ea->arp_spa, (caddr_t) & isaddr, sizeof(isaddr));
375     bcopy((caddr_t) ea->arp_tpa, (caddr_t) & itaddr, sizeof(itaddr));

376     for (ia = in_ifaddr; ia; ia = ia->ia_next)
377         if (ia->ia_ifp == &ac->ac_if) {
378             maybe_ia = ia;
379             if ((itaddr.s_addr == ia->ia_addr.sin_addr.s_addr) ||
380                 (isaddr.s_addr == ia->ia_addr.sin_addr.s_addr))
381                 break;
382         }
383     if (maybe_ia == 0)
384         goto out;
385     myaddr = ia ? ia->ia_addr.sin_addr : maybe_ia->ia_addr.sin_addr;

```

## 358-375

The length of the ether\_arp structure was verified by the caller, so ea is set to point to the received packet. The ARP operation (request or reply) is copied into op but it isn't examined until later in the function. The sender's IP address and target IP address are copied into isaddr and itaddr.

## Look for matching interface and IP address

## 376-382

The linked list of Internet addresses for the host is scanned (the list of `in_ifaddr` structures, [Figure 6.5](#)). Remember that a given interface can have multiple IP addresses. Since the received packet contains a pointer (in the mbuf packet header) to the receiving interface's `ifnet` structure, the only IP addresses considered in the for loop are those associated with the receiving interface. If either the target IP address or the sender's IP address matches one of the IP addresses for the receiving interface, the break terminates the loop.

383-384

If the loop terminates with the variable `maybe_ia` equal to 0, the entire list of configured IP addresses was searched and not one was associated with the received interface. The function jumps to out ([Figure 21.19](#)), where the mbuf is discarded and the function returns. This should only happen if an ARP request is received on an interface that has been initialized but has not been assigned an IP address.

If the for loop terminates having located a receiving interface (maybe\_ia is non-null) but none of its IP addresses matched the sender or target IP address, myaddr is set to the final IP address assigned to the interface. Otherwise (the normal case) myaddr contains the local IP address that matched either the sender or target IP address.

[Figure 21.18](#) shows the next part of the in\_arpinput function, which performs some validation of the packet.

### Figure 21.18. in\_arpinput function: validate received packet.

```

386     if (!bcm((caddr_t) ea->arp_sha, (caddr_t) ac->ac_enaddr,
387             sizeof(ea->arp_sha)))
388         goto out; /* it's from me, ignore it. */
389     if (!bcm((caddr_t) ea->arp_sha, (caddr_t) etherbroadcastaddr,
390             sizeof(ea->arp_sha))) {
391         log(LOG_ERR,
392             "arp: ether address is broadcast for IP address %x!\n",
393             ntohl(isaddr.s_addr));
394         goto out;
395     }
396     if (isaddr.s_addr == myaddr.s_addr) {
397         log(LOG_ERR,
398             "duplicate IP address %x! sent from ethernet address: %s\n",
399             ntohl(isaddr.s_addr), ether_sprintf(ea->arp_sha));
400         itaddr = myaddr;
401         goto reply;
402     }

```

if\_ether.c

if\_ether.c

## Validate sender's hardware address

386-388

If the sender's hardware address equals the hardware address of the interface, the host received a copy of its own request, which is ignored.

389-395

If the sender's hardware address is the Ethernet broadcast address, this is an error. The error is logged and the packet is discarded.

## Check sender's IP address

396-402

If the sender's IP address equals myaddr, then the sender is using the same IP address as this host. This is also an error probably a configuration error by the system administrator on either this host or the sending host. The error is logged and the function jumps to reply ([Figure 21.19](#)),

after setting the target IP address to myaddr (the duplicate address). Notice that this ARP packet could have been destined for some other host on the Ethernet it need not have been sent to this host. Nevertheless, if this form of IP address spoofing is detected, the error is logged and a reply generated.

## Figure 21.19. in\_arpinput function: create a new ARP entry or update existing entry.

```
-----if_ether.c
403     la = arplookup(isaddr.s_addr, itaddr.s_addr == myaddr.s_addr, 0);
404     if (la && (rt = la->la_rt) && (sdl = SDL(rt->rt_gateway))) {
405         if (sdl->sdl_alen &&
406             bcmp((caddr_t) ea->arp_sha, LLADDR(sdl), sdl->sdl_alen))
407             log(LOG_INFO, "arp info overwritten for %x by %s\n",
408                 isaddr.s_addr, ether_sprintf(ea->arp_sha));
409         bcopy((caddr_t) ea->arp_sha, LLADDR(sdl),
410               sdl->sdl_alen = sizeof(ea->arp_sha));
411         if (rt->rt_expire)
412             rt->rt_expire = time.tv_sec + arpt_keep;
413         rt->rt_flags |= RTF_REJECT;
414         la->la_asked = 0;
415         if (la->la_hold) {
416             (*ac->ac_if.if_output) (&ac->ac_if, la->la_hold,
417                                     rt_key(rt), rt);
418             la->la_hold = 0;
419         }
420     }
421     reply:
422     if (op != ARPOP_REQUEST) {
423         out:
424             m_freem(m);
425             return;
426     }
-----if_ether.c
```

Figure 21.19 shows the next part of in\_arpinput.

## Search routing table for match with sender's IP address

403

arplookup searches the ARP cache for the sender's IP address (isaddr). The second argument is 1 if the target IP address equals myaddr (meaning create a new entry if an entry doesn't exist), or 0 otherwise (do not create a new entry). An entry is always created for the sender if this host is the target; otherwise the host is processing a broadcast intended for some other target, so it just looks for an existing entry for the sender. As mentioned earlier, this means that if a host receives an ARP request for itself from another host, an ARP entry is created for that other host on the assumption that, since that host is about to send us a packet, we'll probably send a reply.

The third argument is 0, which means do not look for a proxy ARP entry (described later). The return value is a pointer to an `llinfo_arp` structure, or a null pointer if an

entry is not found or created.

## Update existing entry or fill in new entry

*404*

The code associated with the if statement is executed only if the following three conditions are all true:

- 1. an ARP entry was found or a new ARP entry was successfully created (la is nonnull),**
  - the ARP entry points to a routing table entry (rt), and
  - the rt\_gateway field of the routing table entry points to a sockaddr\_dl structure.

The first condition is false for every broadcast ARP request not directed to this host, from some other host whose IP address is not currently in the routing table.

## Check if sender's hardware addresses

## changed

### 405-408

If the link-level address length (`sdl_alen`) is nonzero (meaning that an existing entry is being referenced and not a new entry that was just created), the link-level address is compared to the sender's hardware address. If they are different, the sender's Ethernet address has changed. This can happen if the sending host is shut down, its Ethernet interface card replaced, and it reboots before the ARP entry times out. While not common, this is a possibility that must be handled. An informational message is logged and the code continues, which will update the hardware address with its new value.

The sender's IP address in the log message should be converted to host byte order. This is a bug.

## Record sender's hardware address

### 409-410

The sender's hardware address is copied into the `sockaddr_dl` structure pointed to by the `rt_gateway` member of the routing table entry. The link-level address length (`sdl_alen`) in the `sockaddr_dl` structure is also set to 6. This assignment of the length field is required if this is a newly created entry ([Exercise 21.3](#)).

## Update newly resolved ARP entry

411-412

When the sender's hardware address is resolved, the following steps occur. If the expiration time is nonzero, it is reset to 20 minutes (`arpt_keep`) in the future. This test exists because the `arp` command can create permanent entries: entries that never time out. These entries are marked with an expiration time of 0. We'll also see in [Figure 21.24](#) that when an ARP request is sent (i.e., for a nonpermanent ARP entry) the expiration time is set to the current time, which is nonzero.

413-414

The RTF\_REJECT flag is cleared and the la\_asked counter is set to 0. We'll see that these last two steps are used in arpresolve to avoid ARP flooding.

## 415-420

If ARP is holding onto an mbuf awaiting ARP resolution of that host's hardware address (the la\_hold pointer), the mbuf is passed to the interface output function. (We show this in [Figure 21.3](#).) Since this mbuf was being held by ARP, the destination address must be on a local Ethernet so the interface output function is ether\_output. This function again calls arpresolve, but the hardware address was just filled in, allowing the mbuf to be queued on the actual device's output queue.

## Finished with ARP reply packets

## 421-426

If the ARP operation is not a request, the received packet is discarded and the

function returns.

The remainder of the function, shown in [Figure 21.20](#), generates a reply to an ARP request. A reply is generated in only two instances:

**1. this host is the target of a request for its hardware address, or**

- this host receives a request for another host's hardware address for which this host has been configured to act as an ARP proxy server.

**Figure 21.20. `in_arpinput` function: form ARP reply and send it.**

```

427     if (ifaddr.s_addr == myaddr.s_addr) {
428         /* I am the target */
429         bcopy((caddr_t) ea->arp_sha, (caddr_t) ea->arp_tha,
430             sizeof(ea->arp_sha));
431         bcopy((caddr_t) ac->ac_enaddr, (caddr_t) ea->arp_sha,
432             sizeof(ea->arp_sha));
433     } else {
434         la = arplookup(itaddr.s_addr, 0, SIN_PROXY);
435         if (la == NULL)
436             goto out;
437         rt = la->la_rt;
438         bcopy((caddr_t) ea->arp_sha, (caddr_t) ea->arp_tha,
439             sizeof(ea->arp_sha));
440         sdl = SDL(rt->rt_gateway);
441         bcopy(LLADDR(sdl), (caddr_t) ea->arp_sha, sizeof(ea->arp_sha));
442     }
443     bcopy((caddr_t) ea->arp_spa, (caddr_t) ea->arp_tpa, sizeof(ea->arp_spa));
444     bcopy((caddr_t) &itaddr, (caddr_t) ea->arp_spa, sizeof(ea->arp_spa));
445     ea->arp_op = htons(ARPOP_REPLY);
446     ea->arp_pro = htons(ETHERTYPE_IP); /* let's be sure! */
447     eh = (struct ether_header *) sa.sa_data;
448     bcopy((caddr_t) ea->arp_tha, (caddr_t) eh->ether_dhost,
449           sizeof(eh->ether_dhost));
450     eh->ether_type = ETHERTYPE_ARP;
451     sa.sa_family = AF_UNSPEC;
452     sa.sa_len = sizeof(sa);
453     (*ac->ac_if.if_output) (&ac->ac_if, m, &sa, (struct rtentry *) 0);
454     return;
455 }

```

if\_ether.c

At this point in the function, an ARP request has been received, but since ARP requests are normally broadcast, the request could be for any system on the Ethernet.

## This host is the target

427-432

If the target IP address equals myaddr, this host is the target of the request. The source hardware address is copied into the target hardware address (i.e., whoever

sent it becomes the target) and the Ethernet address of the interface is copied from the arpcom structure into the source hardware address. The remainder of the ARP reply is constructed after the else clause.

## Check if this host is a proxy server for target

### 433-436

Even if this host is not the target, this host can be configured to be a proxy server for the specified target. arplookup is called again with the create flag set to 0 (the second argument) and the third argument set to SIN\_PROXY. This finds an entry in the routing table only if that entry's SIN\_PROXY flag is set. If an entry is not found (the typical case where this host receives a copy of some other ARP request on the Ethernet), the code at out discards the mbuf and returns.

## Form proxy reply

437-442

To handle a proxy ARP request, the sender's hardware address becomes the target hardware address and the Ethernet address from the ARP entry is copied into the sender hardware address field. This value from the ARP entry can be the Ethernet address of any host on the Ethernet capable of sending IP datagrams to the target IP address. Normally the host providing the proxy ARP service supplies its own Ethernet address, but that's not required. Proxy entries are created by the system administrator using the arp command, with the keyword pub, specifying the target IP address (which becomes the key of the routing table entry) and an Ethernet address to return in the ARP reply.

## **Complete construction of ARP reply packet**

443-444

The remainder of the function completes

the construction of the ARP reply. The sender and target hardware addresses have been filled in. The sender and target IP addresses are now swapped. The target IP address is contained in `itaddr`, which might have been changed if another host was found using this host's IP address ([Figure 21.18](#)).

445-446

The ARP operation is set to `ARPOP_REPLY` and the type of protocol address is set to `ETHERTYPE_IP`. The comment "let's be sure!" is because `arpintr` also calls this function when the type of protocol address is `ETHERTYPE_IPTRAILERS`, but the use of trailer encapsulation is no longer supported.

## Fill in `sockaddr` with Ethernet header

447-452

A `sockaddr` structure is filled in with the 14-byte Ethernet header, as shown in [Figure 21.12](#). The target hardware address

also becomes the Ethernet destination address.

## 453-455

The ARP reply is passed to the interface's output routine and the function returns.

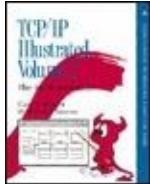
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.9 ARP Timer Functions

ARP entries are normally dynamic they are created when needed and time out automatically. It is also possible for the system administrator to create permanent entries (i.e., no timeout), and the proxy entries we discussed in the previous section are always permanent. Recall from [Figure 21.1](#) and the #define at the end of [Figure 21.10](#) that the rmx\_expire member of the routing metrics structure is used by ARP as a timer.

#### arptimer Function

This function, shown in [Figure 21.21](#), is called every 5 minutes. It goes through all the ARP entries to see if any have expired.

## Figure 21.21. arptimer function: check all ARP timers every 5 minutes.

```
if_ether.c
74 static void
75 arptimer(ignored_arg)
76 void *ignored_arg;
77 {
78     int     s = splnet();
79     struct llinfo_arp *la = llinfo_arp.la_next;
80     timeout(arptimer, (caddr_t) 0, arpt_prune * hz);
81     while (la != &llinfo_arp) {
82         struct rtentry *rt = la->la_rt;
83         la = la->la_next;
84         if (rt->rt_expire && rt->rt_expire <= time.tv_sec)
85             arptfree(la->la_prev); /* timer has expired, clear */
86     }
87     splx(s);
88 }
```

if\_ether.c

## Set next timeout

80

We'll see that the arp\_rtrequest function causes arptimer to be called the first time, and from that point arptimer causes itself to be called 5 minutes (arpt\_prune) in the future.

## Check all ARP entries

81-86

Each entry in the linked list is processed. If the timer is nonzero (it is not a permanent entry) and if the timer has expired, arptfree releases the entry. If rt\_expire is nonzero, it contains a count of the number of seconds since the Unix Epoch when the entry expires.

## arptfree Function

This function, shown in [Figure 21.22](#), is called by arptimer to delete a single entry from the linked list of llinfo\_arp entries.

**Figure 21.22. arptfree function: delete or invalidate an ARP entry.**

---

```

459 static void
460 arptfree(la)
461 struct llinfo_arp *la;
462 {
463     struct rtentry *rt = la->la_rt;
464     struct sockaddr_dl *sdl;
465     if (rt == 0)
466         panic("arptfree");
467     if (rt->rt_refcnt > 0 && (sdl = SDL(rt->rt_gateway)) &&
468         sdl->sdl_family == AF_LINK) {
469         sdl->sdl_alen = 0;
470         la->la_asked = 0;
471         rt->rt_flags &= ~RTF_REJECT;
472         return;
473     }
474     rtrequest(RTM_DELETE, rt_key(rt), (struct sockaddr *) 0, rt_mask(rt),
475               0, (struct rtentry **) 0);
476 }

```

---

if\_ether.c

## Invalidate (don't delete) entries in use

467-473

If the routing table reference count is greater than 0 and the `rt_gateway` member points to a `sockaddr_dl` structure, `arptfree` takes the following steps:

### **1. the link-layer address length is set to 0,**

- the `la_asked` counter is reset to 0, and
- the `RTF_REJECT` flag is cleared.

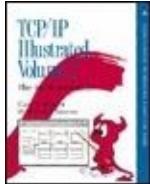
The function then returns. Since the reference count is nonzero, the routing table entry is not deleted. But setting

`sdl_alen` to 0 invalidates the entry, so the next time the entry is used, an ARP request will be generated.

## Delete unreferenced entries

474-475

`rtrequest` deletes the routing table entry, and we'll see in [Section 21.13](#) that it calls `arp_rtrequest`. This latter function frees any mbuf chain held by the ARP entry (the `la_hold` pointer) and deletes the corresponding `linfo_arp` entry.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.10 arpresolve Function

We saw in [Figure 4.16](#) that ether\_output calls arpresolve to obtain the Ethernet address for an IP address. arpresolve returns 1 if the destination Ethernet address is known, allowing ether\_output to queue the IP datagram on the interface's output queue. A return value of 0 means arpresolve does not know the Ethernet address. The datagram is "held" by arpresolve (using the la\_hold member of the IInfo\_arp structure) and an ARP request is sent. If and when an ARP reply is received, in\_arpinput completes the ARP entry and sends the held datagram.

arpresolve must also avoid *ARP flooding*, that is, it must not repeatedly send ARP requests at a high rate when an ARP reply is not received. This can happen when several datagrams are sent to the same unresolved IP address before an ARP reply is received, or when a datagram destined for an unresolved address is fragmented, since each fragment is sent to ether\_output as a separate packet. Section 11.9 of Volume 1 contains an example of ARP flooding caused by fragmentation, and discusses the associated problems. [Figure 21.23](#) shows the first half of arpresolve.

### **Figure 21.23. arpresolve function: find ARP entry if required.**

---

```

252 int
253 arpresolve(ac, rt, m, dst, desten)
254 struct arpcom *ac;
255 struct rtentry *rt;
256 struct mbuf *m;
257 struct sockaddr *dst;
258 u_char *desten;
259 {
260     struct llinfo_arp *la;
261     struct sockaddr_dl *sdl;

262     if (m->m_flags & M_BCAST) { /* broadcast */
263         bcopy((caddr_t) etherbroadcastaddr, (caddr_t) desten,
264               sizeof(etherbroadcastaddr));
265         return (1);
266     }
267     if (m->m_flags & M_MCAST) { /* multicast */
268         ETHER_MAP_IP_MULTICAST(&SIN(dst)->sin_addr, desten);
269         return (1);
270     }
271     if (rt)
272         la = (struct llinfo_arp *) rt->rt_llinfo;
273     else {
274         if (la = arplookup(SIN(dst)->sin_addr.s_addr, 1, 0))
275             rt = la->la_rt;
276     }
277     if (la == 0 || rt == 0) {
278         log(LOG_DEBUG, "arpresolve: can't allocate llinfo");
279         m_free(m);
280         return (0);
281     }

```

---

if\_ether.c

**252-261**

dst is a pointer to a sockaddr\_in containing the destination IP address and desten is an array of 6 bytes that is filled in with the corresponding Ethernet address, if known.

## Handle broadcast and multicast destinations

**262-270**

If the M\_BCAST flag of the mbuf is set, the

destination is filled in with the Ethernet broadcast address and the function returns 1. If the M\_MCAST flag is set, the ETHER\_MAP\_IP\_MULTICAST macro ([Figure 12.6](#)) converts the class D address into the corresponding Ethernet address.

## Get pointer to llinfo\_arp structure

271-276

The destination address is a unicast address. If a pointer to a routing table entry is passed by the caller, la is set to the corresponding llinfo\_arp structure. Otherwise arplookup searches the routing table for the specified IP address. The second argument is 1, telling arplookup to create the entry if it doesn't already exist; the third argument is 0, which means don't look for a proxy ARP entry.

277-281

If either rt or la are null pointers, one of the allocations failed, since arplookup should have created an entry if one didn't

exist. An error message is logged, the packet released, and the function returns 0.

[Figure 21.24](#) contains the last half of arpresolve. It checks whether the ARP entry is still valid, and, if not, sends an ARP request.

**Figure 21.24. arpresolve function: check if ARP entry valid, send ARP request if not.**

```

282     sdl = SDL(rt->rt_gateway);
283     /*
284      * Check the address family and length is valid, the address
285      * is resolved; otherwise, try to resolve.
286      */
287     if ((rt->rt_expire == 0 || rt->rt_expire > time.tv_sec) &&
288         sdl->sdl_family == AF_LINK && sdl->sdl_alen != 0) {
289         bcopy(LLADDR(sdl), desten, sdl->sdl_alen);
290         return 1;
291     }
292     /*
293      * There is an arptab entry, but no ethernet address
294      * response yet. Replace the held mbuf with this
295      * latest one.
296      */
297     if (la->la_hold)
298         m_free(m);
299     la->la_hold = m;

300    if (rt->rt_expire) {
301        rt->rt_flags &= ~RTF_REJECT;
302        if (la->la_asked == 0 || rt->rt_expire != time.tv_sec) {
303            rt->rt_expire = time.tv_sec;
304            if (la->la_asked++ < arp_maxtries)
305                arpwhohas(ac, &(SIN(dst)->sin_addr));
306            else {
307                rt->rt_flags |= RTF_REJECT;
308                rt->rt_expire += arp_tdown;
309                la->la_asked = 0;
310            }
311        }
312    }
313    return (0);
314 }

```

if\_ether.c

## Check ARP entry for validity

282-291

Even though an ARP entry is located, it must be checked for validity. The entry is valid if the following conditions are all true:

- 1. the entry is permanent (the expiration time is 0) or the expiration time is greater than the**

## **current time, and**

- the family of the socket address structure pointed to by `rt_gateway` is `AF_LINK`, and
- the link-level address length (`sdl_alen`) is nonzero.

Recall that `arptfree` invalidated an ARP entry that was still referenced by setting `sdl_alen` to 0. If the entry is valid, the Ethernet address contained in the `sockaddr_dl` is copied into `desten` and the function returns 1.

## **Hold only most recent IP datagram**

292-299

At this point an ARP entry exists but it does not contain a valid Ethernet address. An ARP request must be sent. First the pointer to the mbuf chain is saved in `la_hold`, after releasing any mbuf chain that was already pointed to by `la_hold`. This means that if multiple IP datagrams are sent quickly to a given destination, and an ARP entry does not already exist for the

destination, during the time it takes to send an ARP request and receive a reply only the *last* datagram is held, and all prior ones are discarded. An example that generates this condition is NFS. If NFS sends an 8500-byte IP datagram that is fragmented into six IP fragments, and if all six fragments are sent by `ip_output` to `ether_output` in the time it takes to send an ARP request and receive a reply, the first five fragments are discarded and only the final fragment is sent when the reply is received. This in turn causes an NFS timeout, and a retransmission of all six fragments.

## **Send ARP request but avoid ARP flooding**

*300-314*

RFC 1122 requires ARP to avoid sending ARP requests to a given destination at a high rate when a reply is not received. The technique used by Net/3 to avoid ARP flooding is as follows.

- Net/3 never sends more than one ARP request in any given second to a destination.
- If a reply is not received after five ARP requests (i.e., after about 5 seconds), the RTF\_REJECT flag in the routing table is set and the expiration time is set for 20 seconds in the future. This causes ether\_output to refuse to send IP datagrams to this destination for 20 seconds, returning EHOSTDOWN or EHOSTUNREACH instead ([Figure 4.15](#)).
- After the 20-second pause in ARP requests, arpresolve will send ARP requests to that destination again.

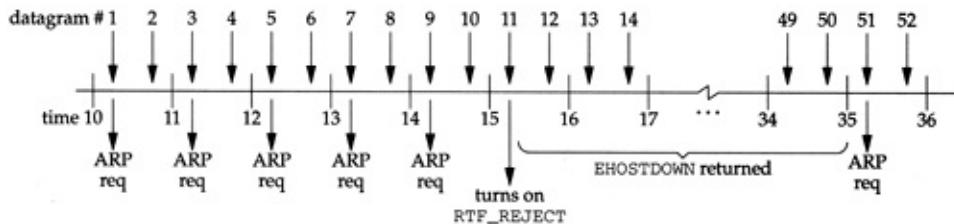
If the expiration time is nonzero (i.e., this is not a permanent entry) the RTF\_REJECT flag is cleared, in case it had been set earlier to avoid flooding. The counter la\_asked counts the number of consecutive times an ARP request has been sent to this destination. If the counter is 0 or if the expiration time does not equal the current time (looking only at the seconds portion of the current time),

an ARP request might be sent. This comparison avoids sending more than one ARP request during any second. The expiration time is then set to the current time in seconds (i.e., the microseconds portion, `time.tv_usec` is ignored).

The counter is compared to the limit of 5 (`arp_maxtries`) and then incremented. If the value was less than 5, `arpwhohas` sends the request. If the request equals 5, however, ARP has reached its limit: the `RTF_REJECT` flag is set, the expiration time is set to 20 seconds in the future, and the counter `la_asked` is reset to 0.

[Figure 21.25](#) shows an example to explain further the algorithm used by `arpresolve` and `ether_output` to avoid ARP flooding.

### Figure 21.25. Algorithm used to avoid ARP flooding.



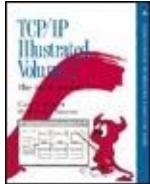
We show 26 seconds of time, labeled 10 through 36. We assume a process is sending an IP datagram every one-half second, causing two datagrams to be sent every second. The datagrams are numbered 1 through 52. We also assume that the destination host is down, so there are no replies to the ARP requests. The following actions take place:

- We assume `la_asked` is 0 when datagram 1 is written by the process. `la_hold` is set to point to datagram 1, `rt_expire` is set to the current time (10), `la_asked` becomes 1, and an ARP request is sent. The function returns 0.
- When datagram 2 is written by the process, datagram 1 is discarded and `la_hold` is set to point to datagram 2. Since `rt_expire` equals the current time (10), nothing else happens (an ARP request is not sent) and the function returns 0.
- When datagram 3 is written, datagram 2 is discarded and `la_hold` is set to point to datagram 3. The current time

(11) does not equal `rt_expire` (10), so `rt_expire` is set to 11. `la_asked` is less than 5, so `la_asked` becomes 2 and an ARP request is sent.

- When datagram 4 is written, datagram 3 is discarded and `la_hold` is set to point to datagram 4. Since `rt_expire` equals the current time (11), nothing else happens and the function returns 0.
- Similar actions occur for datagrams 5 through 10. After datagram 9 causes an ALP request to be sent, `la_asked` is 5.
- When datagram 11 is written, datagram 10 is discarded and `la_hold` is set to point to datagram 11. The current time (15) does not equal `rt_expire` (14), so `rt_expire` is set to 15. `la_asked` is no longer less than 5, so the ARP flooding avoidance algorithm takes place: `RTF_REJECT` flag is set, `rt_expire` is set to 35 (20 seconds in the future), and `la_asked` is reset to 0. The function returns 0.

- When datagram 12 is written, ether\_output notices that the RTF\_REJECT flag is set and that the current time is less than rt\_expire (35) causing EHOSTDOWN to be returned to the sender (normally ip\_output).
- The EHOSTDOWN error is returned for datagrams 13 through 50.
- When datagram 51 is written, even though the RTF\_REJECT flag is set ether\_output does not return the error because the current time (35) is no longer less than rt\_expire (35). arpresolve is called and the entire process starts over again: five ARP requests are sent in 5 seconds, followed by a 20-second pause. This continues until the sending process gives up or the destination host responds to an ARP request.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.11 arplookup Function

arplookup calls the routing function `rtalloc1` to look up an ARP entry in the Internet routing table. We've seen three calls to arplookup:

- 1. from `in_arpinput` to look up and possibly create an entry corresponding to the source IP address of a received ARP packet,**
  - from `in_arpinput` to see if a proxy ARP entry exists for the destination IP address of a received ARP request, and
  - from `arpresolve` to look up or create an

entry corresponding to the destination IP address of a datagram that is about to be sent.

If arplookup succeeds, a pointer is returned to the corresponding `llinfo_arp` structure; otherwise a null pointer is returned.

arplookup has three arguments. The first is the IP address to search for, the second is a flag that is true if a new entry should be created if the entry is not found, and the third is a flag that is true if a proxy ARP entry should be searched for and possibly created.

Proxy ARP entries are handled by defining a different form of the Internet socket address structure, a `sockaddr_inarp` structure, shown in [Figure 21.26](#). This structure is used only by ARP.

**Figure 21.26. `sockaddr_inarp` structure.**

```
if_ether.h
111 struct sockaddr_inarp {
112     u_char    sin_len;           /* sizeof(struct sockaddr_inarp) = 16 */
113     u_char    sin_family;        /* AF_INET */
114     u_short   sin_port;
115     struct in_addr sin_addr;    /* IP address */
116     struct in_addr sin_srcaddr; /* not used */
117     u_short   sin_tos;          /* not used */
118     u_short   sin_other;        /* 0 or SIN_PROXY */
119 };
if_ether.h
```

## 111-119

The first 8 bytes are the same as a `sockaddr_in` structure and the `sin_family` is also set to `AF_INET`. The final 8 bytes, however, are different: the `sin_srcaddr`, `sin_tos`, and `sin_other` members. Of these three, only the final one is used, being set to `SIN_PROXY` (1) if the entry is a proxy entry.

[Figure 21.27](#) shows the `arplookup` function.

**Figure 21.27. `arplookup` function: look up an ARP entry in the routing table.**

```
if_ether.c
480 static struct llinfo_arp *
481 arplookup(addr, create, proxy)
482 u_long addr;
483 int create, proxy;
484 {
485     struct rtentry *rt;
486     static struct sockaddr_inarp sin =
487     {sizeof(sin), AF_INET};

488     sin.sin_addr.s_addr = addr;
489     sin.sin_other = proxy ? SIN_PROXY : 0;
490     rt = rtalloc1((struct sockaddr *)&sin, create);
491     if (rt == 0)
492         return (0);
493     rt->rt_refcnt--;
494     if ((rt->rt_flags & RTF_GATEWAY) || (rt->rt_flags & RTF_LLINFO) == 0 ||
495         rt->rt_gateway->sa_family != AF_LINK) {
496         if (create)
497             log(LOG_DEBUG, "arptnew failed on %x\n", ntohs(addr));
498         return (0);
499     }
500     return ((struct llinfo_arp *) rt->rt_llinfo);
501 }
```

---

```
if_ether.c
```

## Initialize sockaddr\_inarp to look up

480-489

The sin\_addr member is set to the IP address that is being looked up. The sin\_other member is set to SIN\_PROXY if the proxy argument is nonzero, or 0 otherwise.

## Look up entry in routing table

490-492

rtalloc1 looks up the IP address in the Internet routing table, creating a new

entry if the create argument is nonzero. If the entry is not found, the function returns 0 (a null pointer).

## Decrement routing table reference count

493

If the entry is found, the reference count for the routing table entry is decremented. This is because ARP is not considered to "hold onto" a routing table entry like the transport layers, so the increment of `rt_refcnt` that was done by the routing table lookup is undone here by ARP.

494-499

If the `RTF_GATEWAY` flag is set, or the `RTF_LLINFO` flag is not set, or the address family of the socket address structure pointed to by `rt_gateway` is not `AF_LINK`, something is wrong and a null pointer is returned. If the entry was created this way, a log message is created.

The log message with the function name `arptnew` refers to the older Net/2

function that created ARP entries.

If rtalloc1 creates a new entry because the matching entry had the RTF\_CLONING flag set, the function arp\_rtrequest (which we describe in [Section 21.13](#)) is also called by rtrequest.

---

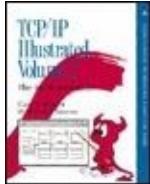
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.12 Proxy ARP

Net/3 supports proxy ARP, as we saw in the previous section. Two different types of proxy ARP entries can be added to the routing table. Both are added with the arp command, specifying the pub option. Adding a proxy ARP entry always causes a gratuitous ARP request to be issued by arp\_rtrequest (Figure 21.28) because the RTF\_ANNOUNCE flag is set when the entry is created.

**Figure 21.28. arp\_rtrequest function:  
RTM\_ADD command.**

```

92 void
93 arp_rtrequest(req, rt, sa)
94 int     req;
95 struct rtentry *rt;
96 struct sockaddr *sa;
97 {
98     struct sockaddr *gate = rt->rt_gateway;
99     struct llinfo_arp *la = (struct llinfo_arp *) rt->rt_llinfo;
100    static struct sockaddr_dl null_sdl =
101        {sizeof(null_sdl), AF_LINK};

102    if (!arpinit_done) {
103        arpinit_done = 1;
104        timeout(arptimer, (caddr_t) 0, hz);
105    }
106    if (rt->rt_flags & RTF_GATEWAY)
107        return;
108    switch (req) {

109    case RTM_ADD:
110        /*
111         * XXX: If this is a manually added route to interface
112         * such as older version of routed or gated might provide,
113         * restore cloning bit.
114         */
115        if ((rt->rt_flags & RTF_HOST) == 0 &&
116            SIN(rt->rt_mask(rt))->sin_addr.s_addr != 0xffffffff)
117            rt->rt_flags |= RTF_CLONING;
118        if (rt->rt_flags & RTF_CLONING) {
119            /*
120             * Case 1: This route should come from a route to iface.
121             */
122            rt_setgate(rt, rt_key(rt),
123                       (struct sockaddr *) &null_sdl);
124            gate = rt->rt_gateway;
125            SDL(gate)->sdl_type = rt->rt_ifp->if_type;
126            SDL(gate)->sdl_index = rt->rt_ifp->if_index;
127            rt->rt_expire = time.tv_sec;
128            break;
129        }
130        /* Announce a new entry if requested. */
131        if (rt->rt_flags & RTF_ANNOUNCE)
132            arprequest((struct arpcmd *) rt->rt_ifp,
133                        &SIN(rt_key(rt))->sin_addr.s_addr,
134                        &SIN(rt_key(rt))->sin_addr.s_addr,
135                        (u_char *) LLADDR(SDL(gate)));
136    /* FALLTHROUGH */

```

The first type of proxy ARP entry allows an IP address for a host on an attached network to be entered into the ARP cache. Any Ethernet address can be assigned to the entry. These entries are added to the routing table with an explicit mask of 0xffffffff. The purpose of this mask is to allow the call to `rtalloc1` in [Figure 21.27](#) to

match this entry, even if the SIN\_PROXY flag is set in the socket address structure of the search key. This in turn allows the call to arplookup from [Figure 21.20](#) to match this entry when a search is made for the target address with the SIN\_PROXY flag set.

This type of entry can be used if a host H1 that doesn't implement ARP is on an attached network. The host with the proxy entry answers all ARP requests for H1's hardware address, supplying the Ethernet address that was specified when the proxy entry was created (i.e., the Ethernet address of H1). These entries are output with the notation "published" by the arp -a command.

The second type of proxy ARP entry is for a host for which a routing table entry already exists. The kernel creates another routing table entry for the destination, with this new entry containing the link-layer information (i.e., the Ethernet address). The SIN\_PROXY flag is set in the sin\_other member of the sockaddr\_inarp structure ([Figure 21.26](#)) in the new routing

table entry. Recall that routing table searches compare 12 bytes of the Internet socket address structure ([Figure 18.39](#)). This use of the SIN\_PROXY flag is the only time the final 8 bytes of the structure are nonzero. When arplookup specifies the SIN\_PROXY value in the sin\_other member of the structure passed to rtalloc1, the only entries in the routing table that will match are ones that also have the SIN\_PROXY flag set.

This type of entry normally specifies the Ethernet address of the host acting as the proxy server. If the proxy entry was created for a host HD, the sequence of steps is as follows.

**1. The proxy server receives a broadcast ARP request for HD's hardware address from some other host HS. The host HS thinks HD is on the local network.**

- The proxy server responds, supplying its own Ethernet address.
- HS sends the datagram with a

destination IP address of HD to the proxy server's Ethernet address.

- The proxy server receives the datagram for HD and forwards it, using the normal routing table entry for HD.

This type of entry was used on the router netb in the example in Section 4.6 of Volume 1. These entries are output by the arp -a command with the notation "published (proxy only)."



## Chapter 21. ARP: Address Resolution F

### 21.13 arp\_rtrequest Function

Figure 21.3 provides an overview of the relationship between the ARP functions and the routing functions. We've encountered two calls to the routing table from the ARP functions.

1. **arplookup calls rtalloc1 to look up an AII and possibly create a new entry if a matching entry is found.**

If a matching entry is found in the routing table and the RTF\_CLONING flag is not set (i.e., no matching entry for the destination host), a pointer to the matching entry is returned. If the RTF\_CLONING bit is set, rtalloc1 calls arp\_rtrequest with a command of RTM\_RESOLVE. This is how the entries for 140.252.13.34 and 140.252.13.34 in Figure 18.2 were created.

## were cloned from the entry for 140.252

- arptfree calls rtrequest with a command of RT to delete an entry from the routing table that corresponds to an ARP entry.

Additionally, the arp command manipulates the cache by sending and receiving routing messages via a routing socket. The arp command issues routine messages with commands of RTM\_ADD, RTM\_DEL, and RTM\_GET. The first two commands cause rtalloc1 to be called and the third causes rtalloc1 to be called.

Finally, when an Ethernet device driver has an IP address assigned to the interface, rtinit adds a route to the network. This causes rtrequest to be called with a command of RTM\_ADD and with the flags of RTF\_UP and RTF\_CLONING. This is how the entry for 140.252 was created.

As described in [Chapter 19](#), each ifaddr structure contains a pointer to a function (the ifa\_rtrequest member) that is automatically called when a routing table entry is added or deleted for that interface. We saw in [Figure 6.17](#) that in\_ifinit sets this pointer to the arp\_rtrequest for all Ethernet devices. Therefore whenever the routing functions are called to add or delete a routing table entry for ARP, arp\_rtrequest is called.

The purpose of this function is to do whatever t initialization or cleanup is required above and b what the generic routing table functions perform example, this is where a new llinfo\_arp struct is allocated and initialized whenever a new ARP entry is created. In a similar way, the llinfo\_arp structure is deleted by this function after the generic routine have completed processing an RTM\_DELETE command.

[Figure 21.28](#) shows the first part of the arp\_rtroute function.

## Initialize ARP timeout function

92-105

The first time arp\_rtrequest is called (when the Ethernet interface is assigned an IP address during system initialization), the timeout function schedules the function arptimer to be called in 1 second. This ARP timer code runs every 5 minutes, since it always calls timeout.

## Ignore indirect routes

106-107

If the RTF\_GATEWAY flag is set, the function reflag indicates an indirect routing table entry and entries are direct routes.

108

The remainder of the function is a switch with three cases: RTM\_ADD, RTM\_RESOLVE, and RTM\_DELETE (the latter two are shown in figures that follow.)

## RTM\_ADD command

109

The first case for RTM\_ADD is invoked by either command manually creating an ARP entry or by Ethernet interface being assigned an IP address (Figure 21.3).

## Backward compatibility

110-117

If the RTF\_HOST flag is cleared, this routing table entry has an associated mask (i.e., it is a network route or host route). If that mask is not all one bits, the entry is really a route to an interface, so the

RTF\_CLONING flag is set. As the comment indicates, this is for backward compatibility with older version routing daemons. Also, the command

```
route add -net 224.0.0.0 -interf
```

that is in the file /etc/netstart creates the entry for the network shown in [Figure 18.2](#) that has the RTF\_CLONING flag set.

## Initialize entry for network route to interface

118-126

If the RTF\_CLONING flag is set (which it is for Ethernet interfaces), this entry is probably being initialized by rtinit. rt\_setgate allocates space for a sockaddr structure, which is pointed to by the rt\_gateway member. This data-link socket address structure is the one associated with the routing table entry for 140. It is initialized in [Figure 21.1](#). The sdl\_len and sdl\_family members are initialized from the static definition of null\_sdl at the beginning of the function, and the sdl\_type (probably IFT\_ETHER) and sdl\_index members are copied from the interface's ifnet structure. This structure never contains an Ethernet address and the sdl\_alen member

127-128

Finally, the expiration time is set to the current which is simply the time the entry was created, break causes the function to return. For entries system initialization, their rmx\_expire value is 0 which the system was bootstrapped. Notice in Figure 18.2 that this routing table entry does not have an associated Iinfo\_arp structure, so it is never processed by arp. Nevertheless this sockaddr\_dl structure is used in the rt\_gateway structure for the entry that is cloned for host-specific entries on this Ethernet, it is copied into the rtrequest when the newly cloned entries are created via the RTM\_RESOLVE command. Also, the netstat command prints the sdl\_index value as link#n, as we see in Figure 18.2.

## Send gratuitous ARP request

130-135

If the RTF\_ANNOUNCE flag is set, this entry is created by the arp command with the pub option. This option has two ramifications: (1) the SIN\_PROUTED flag must be set in the sin\_other member of the sockaddr\_dl structure, and (2) the RTF\_ANNOUNCE flag will be cleared. Since the RTF\_ANNOUNCE flag is set, arprequest will

broadcasts a gratuitous ARP request. Notice that the second and third arguments are the same, which makes the sender IP address to equal the target IP address in the ARP request.

136

The code falls through to the case for the RTM\_F command.

[Figure 21.29](#) shows the next part of the arp\_rtr function, which handles the RTM\_RESOLVE command. This command is issued when rtalloc1 matches with the RTF\_CLONING flag set and its second argument is nonzero (the create argument to arplookup). An Iinfo\_arp structure must be allocated and initialized.

**Figure 21.29. arp\_rtrequest function: RTM\_F command.**

```

137     case RTM_RESOLVE:
138         if (gate->sa_family != AF_LINK ||
139             gate->sa_len < sizeof(null_sdl)) {
140             log(LOG_DEBUG, "arp_rtrequest: bad gateway value");
141             break;
142         }
143         SDL(gate)->sdl_type = rt->rt_ifp->if_type;
144         SDL(gate)->sdl_index = rt->rt_ifp->if_index;
145         if (la != 0)
146             break;           /* This happens on a route change */
147         /*
148          * Case 2: This route may come from cloning, or a manual route
149          * add with a LL address.
150          */
151         R_Malloc(la, struct llinfo_arp *, sizeof(*la));
152         rt->rt_llinfo = (caddr_t) la;
153         if (la == 0) {
154             log(LOG_DEBUG, "arp_rtrequest: malloc failed\n");
155             break;
156         }
157         arp_inuse++, arp_allocated++;
158         Bzero(la, sizeof(*la));

159         la->la_rt = rt;
160         rt->rt_flags |= RTF_LLINFO;
161         insque(la, &llinfo_arp);

162         if (SIN(rt_key(rt))->sin_addr.s_addr ==
163             (IA_SIN(rt->rt_ifa))->sin_addr.s_addr) {
164             /*
165              * This test used to be
166              * if (loif.if_flags & IFF_UP)
167              * It allowed local traffic to be forced
168              * through the hardware by configuring the loopback down.
169              * However, it causes problems during network configuration
170              * for boards that can't receive packets they send.
171              * It is now necessary to clear "useloopback" and remove
172              * the route to force traffic out to the hardware.
173              */
174             rt->rt_expire = 0;
175             Bcopy((struct arpcom *) rt->rt_ifp)->ac_enaddr,
176                   LLADDR(SDL(gate)), SDL(gate)->sdl_alen = 6);
177             if (useloopback)
178                 rt->rt_ifp = &loif;

179         }
180         break;

```

---

if\_ether.c

## Verify sockaddr\_dl Structure

137-144

The family and length of the sockaddr\_dl struct pointed to by the rt\_gateway pointer are verified. The interface type (probably IFT\_ETHER) and index

copied into the new sockaddr\_dl structure.

## Handle route changes

145-146

Normally the routing table entry is new and does not point to an llinfo\_arp structure. If the la pointer is nonnull, however, arp\_rtrequest was called when changed for an existing routing table entry. Since the llinfo\_arp structure is already allocated, the brcm function to return.

## Initialize llinfo\_arp structure

147-158

An llinfo\_arp structure is allocated and its pointer stored in the rt\_llinfo pointer of the routing table entry. The two statistics arp\_inuse and arp\_allocated are incremented and the llinfo\_arp structure is set. The la\_hold field is set to a null pointer and la\_asked to 0.

159-161

The rt pointer is stored in the llinfo\_arp structure and the RTF\_LLINFO flag is set. In [Figure 18.2](#) we see the structure.

three routing table entries created by ARP, 140.252.13.33, 140.252.13.34, and 140.252.1 have the L flag enabled, as does the entry for 2. Recall that the arp program looks only for entries with this flag ([Figure 19.36](#)). Finally the new structure is added to the front of the linked list of llinfo\_arp structures by insque.

The ARP entry has been created: rtrequest creates a routing table entry (often cloning a network-specific entry for the Ethernet) and arp\_rtrequest allocates an interface and initializes an llinfo\_arp structure. All that remains is to send the ARP request to be broadcast so that an ARP reply can be received in the host's Ethernet address. In the common case of events, arp\_rtrequest is called because arpresolve has already called arplookup (the intermediate sequence of calls can be followed in [Figure 21.3](#)). When control returns to arpresolve, it broadcasts the ARP request.

## Handle local host specially

162-173

This portion of code is a special test that is new to 4.4BSD (although the comment is left over from earlier releases). It creates the rightmost routing table entry shown in [Figure 21.1](#) with a key consisting of the local host's IP address.

address (140.252.13.35). The if test checks whether the routing table key equals the IP address of the interface, so, the entry that was just created (probably associated with the interface entry) refers to the local host.

## Make entry permanent and set Ethernet address

174-176

The expiration time is set to 0, making the entry permanent. It will never time out. The Ethernet address is copied from the arpcom structure of the interface, which is the sockaddr\_dl structure pointed to by the rt\_gateway member.

## Set interface pointer to loopback interface

177-178

If the global useloopback is nonzero (it defaults to 0), the interface pointer in the routing table entry is checked to see if it points to the loopback interface. This means that datagrams sent to the host's own IP address are sent via the loopback interface instead. Prior to 4.4BSD, the connection from the host's own IP address to the loopback interface was established using a command of the form:

```
route add 140.252.13.35 127.0.0.
```

in the /etc/netstart file. Although this still work 4.4BSD, it is unnecessary because the code we looked at creates an equivalent route automatically the first time an IP datagram is sent to the host's broadcast address. Also realize that this piece of code is executed only once per interface. Once the routing table contains the permanent ARP entry are created, they do not need to be created again so another RTM\_RESOLVE for this IP address will be ignored.

The final part of arp\_rtrequest, shown in Figure 21.30, handles the RTM\_DELETE request. From Figure 21.29 see that this command can be generated from the route command, to delete an entry manually, and from the arptfree function, when an ARP entry times out.

**Figure 21.30. arp\_rtrequest function: RTM\_DELETE command.**

```
if_ether.c
181     case RTM_DELETE:
182         if (la == 0)
183             break;
184         arp_inuse--;
185         remque(la);
186         rt->rt_llinfo = 0;
187         rt->rt_flags &= ~RTF_LLINFO;
188         if (la->la_hold)
189             m_free(m_free(la->la_hold));
190         Free((caddr_t) la);
191     }
192 }
```

if\_ether.c

## Verify la pointer

182-183

The la pointer should always be nonnull (that is routing table entry should always point to an Ili structure); otherwise the break causes the function to return.

## Delete Ilinfo\_arp structure

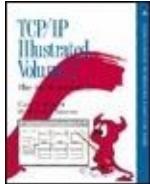
184-190

The arp\_inuse statistic is decremented and the structure is removed from the doubly linked list remque. The rt\_llinfo pointer is set to 0 and the RTF\_LLINFO flag is cleared. If an mbuf is held by this entry (i.e., an ARP request is outstanding), that is released. Finally the Ilinfo\_arp structure is released.

Notice that the switch statement does not provide a default case and does not provide a case for the RTM\_GET command. This is because the RTM\_GET command issued by the arp program is handled by the route\_output function, and rtrequest is not used. Also, the call to rtalloc1 that we show in [Figure 182](#) which is caused by an RTM\_GET command, specifies

second argument of 0; therefore `rtalloc1` does not request in this case.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.14 ARP and Multicasting

If an IP datagram is destined for a multicast group, ip\_output checks whether the process has assigned a specific interface to the socket ([Figure 12.40](#)), and if so, the datagram is sent out that interface. Otherwise, ip\_output selects the outgoing interface using the normal IP routing table ([Figure 8.24](#)). Therefore, on a system with more than one multicast-capable interface, the IP routing table specifies the default interface for each multicast group.

We saw in [Figure 18.2](#) that an entry was created in our routing table for the

224.0.0.0 network and since that entry has its "clone" flag set, all multicast groups starting with 224 had the associated interface (le0) as its default. Additional routing table entries can be created for the other multicast groups (the ones beginning with 225-239), or specific entries can be created for particular multicast groups to assign an explicit default. For example, a routing table entry could be created for 224.0.1.1 (the network time protocol) with an interface that differs from the interface for 224.0.0.0. If an entry for a multicast group does not exist in the routing table, and the process doesn't specify an interface with the IP\_MULTICAST\_IF socket option, the default interface for the group becomes the interface associated with the "default" route in the table. In [Figure 18.2](#) the entry for 224.0.0.0 isn't really needed, since both it and the default route use the interface le0.

Once the interface is selected, if the interface is an Ethernet, arpresolve is called to convert the multicast group address into its corresponding Ethernet

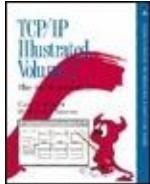
address. In [Figure 21.23](#) this was done by invoking the macro `ETHER_MAP_IP_MULTICAST`. Since this simple macro logically ORs the low-order 23 bits of the multicast group with a constant ([Figure 12.6](#)), an ARP request-reply is not required and the mapping does not need to go into the ARP cache. The macro is just invoked each time the conversion is required.

Multicast group addresses appear in the Net/3 ARP cache if the multicast group is cloned from another entry, as we saw in [Figure 21.5](#). This is because these entries have the `RTF_LLINFO` flag set. These are not true ARP entries because they do not require an ARP requestreply, and they do not have an associated link-layer address, since the mapping is done when needed by the `ETHER_MAP_IP_MULTICAST` macro.

The timeout of the ARP entries for these multicast group addresses is different from normal ARP entries. When a routing table entry is created for a multicast group, such as the entry for 224.0.0.1 in [Figure 18.2](#), `rtrequest` copies the `rt_metrics` structure

from the entry being cloned (Figure 19.9). We mentioned with Figure 21.28 that the network entry has an rmx\_expire value of the time the RTM\_ADD command was executed, normally the time the system was initialized. The new entry for 224.0.0.1 has this same expiration time.

This means the ARP entry for a multicast group such as 224.0.0.1 expires the next time arptimer executes, because its expiration time is always in the past. The entry is created again the next time it is looked up in the routing table.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 21. ARP: Address Resolution Protocol

### 21.15 Summary

ARP provides the dynamic mapping between IP addresses and hardware addresses. This chapter has examined an implementation of ARP that maps IP addresses to Ethernet addresses.

The Net/3 implementation is a major change from previous BSD releases. The ARP information is now stored in various structures: the routing table, a data-link socket address structure, and an `llinfo_arp` structure. [Figure 21.1](#) shows the relationships between all the structures.

Sending an ARP request is simple: the

appropriate fields are filled in and the request is sent as a broadcast. Processing a received request is more complicated because each host receives *all* broadcast ARP requests. Besides responding to requests for one of the host's IP addresses, `in_arpinput` also checks that some other host isn't using the host's IP address. Since all ARP requests contain the sender's IP and hardware addresses, any host on the Ethernet can use this information to update an existing ARP entry for the sender.

ARP flooding can be a problem on a LAN and Net/3 is the first BSD release to handle this. A maximum of one ARP request per second is sent to any given destination, and after five consecutive requests without a reply, a 20-second pause occurs before another ARP request is sent to that destination.

## Exercises

What assumption is made in the

**21.1** assignment of the local variable ac  
in Figure 21.17?

If we ping the broadcast address of  
the local Ethernet and then execute  
**21.2** arp -a, we see that this causes the  
ARP cache to be filled with entries  
for almost every other host on the  
local Ethernet. Why?

Follow through the code and  
**21.3** explain why the assignment of 6 to  
sdl\_alen is required in Figure 21.19.

With the separate ARP table in  
Net/2, independent of the routing  
table, each time arpresolve was  
**21.4** called, a search was made of the  
ARP table. Compare this to the  
Net/3 approach. Which is more  
efficient?

The ARP code in Net/2 explicitly set  
a timeout of 3 minutes for an

incomplete entry in the ARP cache,  
**21.5** that is, for an entry that is awaiting an ARP reply. We've never explicitly said how Net/3 handles this timeout. When does Net/3 time out an incomplete ARP entry?

What changes in the avoidance of ARP flooding when a Net/3 system  
**21.6** is acting as a router and the packets that cause the flooding are from some other host?

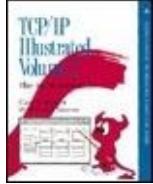
What are the values of the four  
**21.7** `rmx_expire` variables shown in [Figure 21.1](#)? Where in the code are the values set?

What change would be required to the code in this chapter to cause an  
**21.8** ARP entry to be created for every host that broadcasts an ARP request?

To verify the example in [Figure 21.25](#) the authors ran the sock program from Appendix C of Volume 1, writing a UDP datagram every 500 ms to a nonexistent host **21.9** on the local Ethernet. (The -p option of the program was modified to allow millisecond waits.) But only 10 UDP datagrams were sent without an error, instead of the 11 shown in [Figure 21.25](#), before the first EHOSTDOWN error was returned. Why?

Modify ARP to hold onto *all* packets for a destination, awaiting an ARP reply, instead of just the most recent one. What are the **21.10** implications of this change? Should there be a limit, as there is for each interface's output queue? Are any changes required to the data structures?

Top



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 22. Protocol Control Blocks

[Section 22.1. Introduction](#)

[Section 22.2. Code Introduction](#)

[Section 22.3. inpcb Structure](#)

[Section 22.4. in\\_pcalloc and  
in\\_pcbdetach Functions](#)

[Section 22.5. Binding, Connecting, and  
Demultiplexing](#)

[Section 22.6. in\\_pcblayout Function](#)

[Section 22.7. in\\_pcbbind Function](#)

[Section 22.8. in\\_pcconnect Function](#)

[Section 22.9. in\\_pcbs disconnect  
Function](#)

[Section 22.10. in\\_setsockaddr and  
in\\_setpeeraddr Functions](#)

## Section 22.11. in\_pcnotify, in\_rtchange, and in\_losing Functions

## Section 22.12. Implementation Refinements

## Section 22.13. Summary

---

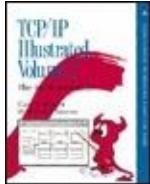
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.1 Introduction

Protocol control blocks (PCBs) are used at the protocol layer to hold the various pieces of information required for each UDP or TCP socket. The Internet protocols maintain *Internet protocol control blocks* and *TCP control blocks*. Since UDP is connectionless, everything it needs for an end point is found in the Internet PCB; there are no UDP control blocks.

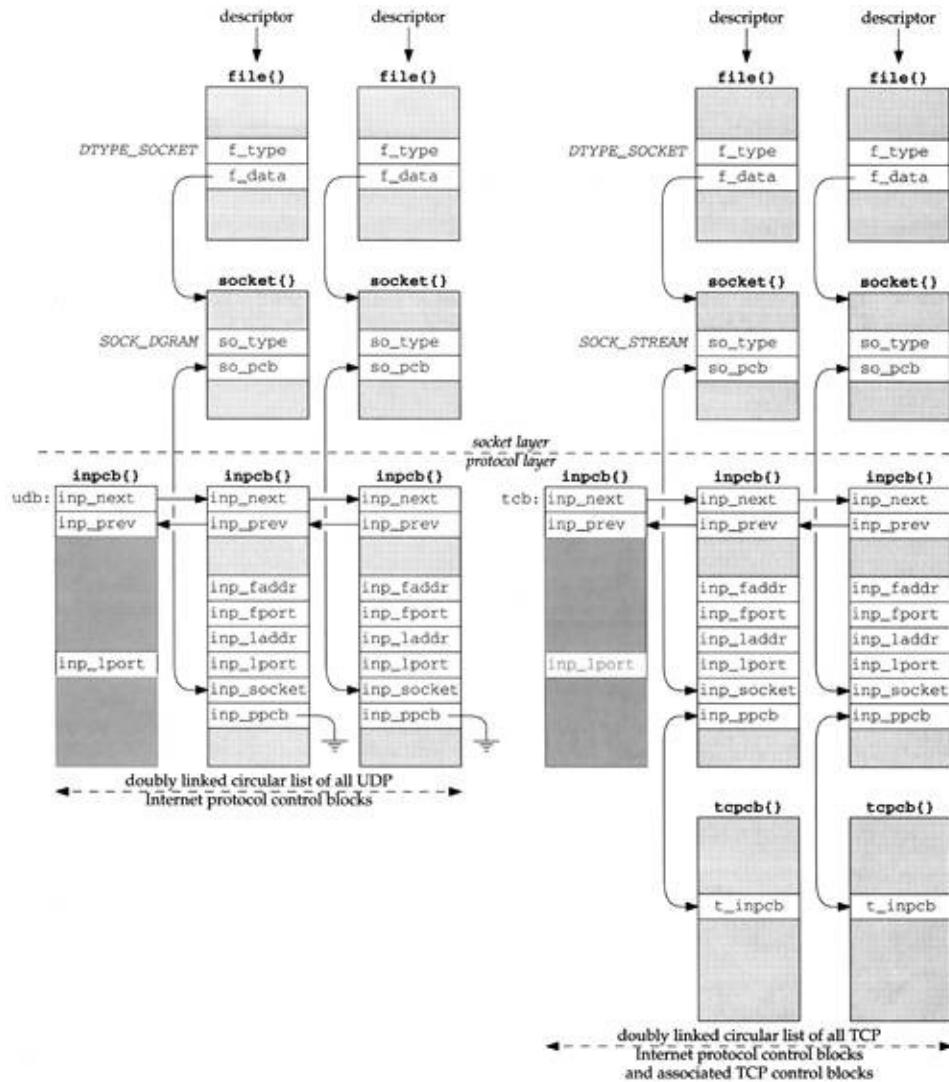
The Internet PCB contains the information common to all UDP and TCP end points: foreign and local IP addresses, foreign and local port numbers, IP header prototype, IP options to use for this end point, and a

pointer to the routing table entry for the destination of this end point. The TCP control block contains all of the state information that TCP maintains for each connection: sequence numbers in both directions, window sizes, retransmission timers, and the like.

In this chapter we describe the Internet PCBs used in Net/3, saving TCP's control blocks until we describe TCP in detail. We examine the numerous functions that operate on Internet PCBs, since we'll encounter them when we describe UDP and TCP. Most of the functions begin with the six characters `in_pcb`.

[Figure 22.1](#) summarizes the protocol control blocks that we describe and their relationship to the file and socket structures. There are numerous points to consider in this figure.

**Figure 22.1. Internet protocol control blocks and their relationship to other structures.**



- When a socket is created by either socket or accept, the socket layer creates a file structure and a socket structure. The file type is DTTYPE\_SOCKET and the socket type is SOCK\_DGRAM for UDP end points or SOCK\_STREAM for TCP end points.
- The protocol layer is then called. UDP

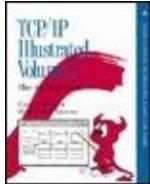
creates an Internet PCB (an `inpcb` structure) and links it to the socket structure: the `so_pcb` member points to the `inpcb` structure and the `inp_socket` member points to the socket structure.

- TCP does the same and also creates its own control block (a `tcpcb` structure) and links it to the `inpcb` using the `inp_ppcb` and `t_inpcb` pointers. In the two UDP `inpcbs` the `inp_ppcb` member is a null pointer, since UDP does not maintain its own control block.
- The four other members of the `inpcb` structure that we show, `inp_faddr` through `inp_lport`, form the socket pair for this end point: the foreign IP address and port number along with the local IP address and port number.
- Both UDP and TCP maintain a doubly linked list of all their Internet PCBs, using the `inp_next` and `inp_prev` pointers. They allocate a global `inpcb` structure as the head of their list (named `udb` and `tcb`) and only use three members in the structure: the

next and previous pointers, and the local port number. This latter member contains the next ephemeral port number to use for this protocol.

The Internet PCB is a transport layer data structure. It is used by TCP, UDP, and raw IP, but not by IP, ICMP, or IGMP.

We haven't described raw IP yet, but it too uses Internet PCBs. Unlike TCP and UDP, raw IP does not use the port number members in the PCB, and raw IP uses only two of the functions that we describe in this chapter: `in_pcbaalloc` to allocate a PCB, and `in_pcbadetach` to release a PCB. We return to raw IP in [Chapter 32](#).



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

## 22.2 Code Introduction

All the PCB functions are in a single C file and a single header contains the definitions, as shown in [Figure 22.2](#).

**Figure 22.2. Files discussed in this chapter.**

File	Description
netinet/in_pcbs.h	inpcb structure definition
netinet/in_pcbs.c	PCB functions

## Global Variables

One global variable is introduced in this

chapter, which is shown in [Figure 22.3](#).

## Figure 22.3. Global variable introduced in this chapter.

Variable	Datatype	Description
zeroin_addr	struct in_addr	32-bit IP address of all zero bits

## Statistics

Internet PCBs and TCP PCBs are both allocated by the kernel's malloc function with a type of M\_PCB. This is just one of the approximately 60 different types of memory allocated by the kernel. Mbufs, for example, are allocated with a type of M\_BUF, and socket structures are allocated with a type of M\_SOCKET.

Since the kernel can keep counters of the different types of memory buffers that are allocated, various statistics on the number of PCBs can be maintained. The command vmstat -m shows the kernel's memory allocation statistics and the netstat -m command shows the mbuf allocation

statistics.

---

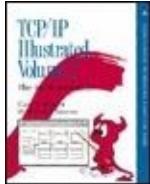
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.3 inpcb Structure

Figure 22.4 shows the definition of the inpcb structure. It is not a big structure, and occupies only 84 bytes.

#### Figure 22.4. inpcb structure.

```
in_pcbs.h
42 struct inpcb {
43     struct inpcb *inp_next, *inp_prev; /* doubly linked list */
44     struct inpcb *inp_head;          /* pointer back to chain of inpcb's for
45                                     this protocol */
46     struct in_addr inp_faddr;      /* foreign IP address */
47     u_short inp_fport;            /* foreign port# */
48     struct in_addr inp_laddr;      /* local IP address */
49     u_short inp_lport;            /* local port# */
50     struct socket *inp_socket;    /* back pointer to socket */
51     caddr_t inp_ppcb;            /* pointer to per-protocol PCB */
52     struct route inp_route;       /* placeholder for routing entry */
53     int     inp_flags;             /* generic IP/datagram flags */
54     struct ip inp_ip;             /* header prototype; should have more */
55     struct mbuf *inp_options;     /* IP options */
56     struct ip_moptions *inp_moptions; /* IP multicast options */
57 };
```

43-45

inp\_next and inp\_prev form the doubly linked list of all PCBs for UDP and TCP. Additionally, each PCB has a pointer to the head of the protocol's linked list (inp\_head). For PCBs on the UDP list, inp\_head always points to udb ([Figure 22.1](#)); for PCBs on the TCP list, this pointer always points to tcb.

46-49

The next four members, inp\_faddr, inp\_fport, inp\_laddr, and inp\_lport, contain the socket pair for this IP end point: the foreign IP address and port number and the local IP address and port number. These four values are maintained in the PCB in network byte order, not host byte order.

The Internet PCB is used by both transport layers, TCP and UDP. While it makes sense to store the local and foreign IP addresses in this structure, the port numbers really don't belong here. The definition of a port number

and its size are specified by each transport layer and could differ between different transport layers. This problem was identified in [Partridge 1987], where 8-bit port numbers were used in version 1 of RDP, which required reimplementing several standard kernel routines to use 8-bit port numbers. Version 2 of RDP [Partridge and Hinden 1990] uses 16-bit port numbers. The port numbers really belong in a transport-specific control block, such as TCP's tcpcb. A new UDP-specific PCB would then be required. While doable, this would complicate some of the routines we'll examine shortly.

50-51

inp\_socket is a pointer to the socket structure for this PCB and inp\_ppcb is a pointer to an optional transport-specific control block for this PCB. We saw in Figure 22.1 that the inp\_ppcb pointer is used with TCP to point to the corresponding tcpcb, but is not used by UDP. The link between the socket and inpcb is two way because sometimes the

kernel starts at the socket layer and needs to find the corresponding Internet PCB (e.g., user output), and sometimes the kernel starts at the PCB and needs to locate the corresponding socket structure (e.g., processing a received IP datagram).

52

If IP has a route to the foreign address, it is stored in the `inp_route` entry. We'll see that when an ICMP redirect message is received, all Internet PCBs are scanned and all those with a foreign IP address that matches the redirected IP address have their `inp_route` entry marked as invalid. This forces IP to find a new route to the foreign address the next time the PCB is used for output.

53

Various flags are stored in the `inp_flags` member. [Figure 22.5](#) lists the individual flags.

**Figure 22.5. `inp_flags` values.**

inp_flags	Description
<code>INP_HDRINCL</code>	process supplies entire IP header (raw socket only)
<code>INP_RECVOPTS</code>	receive incoming IP options as control information (UDP only, not implemented)
<code>INP_RECVRETOPTS</code>	receive IP options for reply as control information (UDP only, not implemented)
<code>INP_RECVDSTADDR</code>	receive IP destination address as control information (UDP only)
<code>INP_CONTROLOPTS</code>	<code>INP_RECVOPTS</code>   <code>INP_RECVRETOPTS</code>   <code>INP_RECVDSTADDR</code>

54

A copy of an IP header is maintained in the PCB but only two members are used, the TOS and TTL. The TOS is initialized to 0 (normal service) and the TTL is initialized by the transport layer. We'll see that TCP and UDP both default the TTL to 64. A process can change these defaults using the `IP_TOS` or `IP_TTL` socket options, and the new value is recorded in the `inpcb.inp_ip` structure. This structure is then used by TCP and UDP as the prototype IP header when sending IP datagrams.

55-56

A process can set the IP options for outgoing datagrams with the `IP_OPTIONS` socket option. A copy of the caller's options are stored in an mbuf by the function `ip_pcbopts` and a pointer to that mbuf is stored in the `inp_options` member.

Each time TCP or UDP calls the ip\_output function, a pointer to these IP options is passed for IP to insert into the outgoing IP datagram. Similarly, a pointer to a copy of the user's IP multicast options is maintained in the inp\_moptions member.

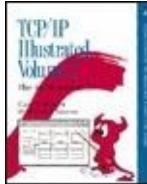
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.4 in\_pcbaalloc and in\_pcbadetach Functions

An Internet PCB is allocated by TCP, UDP, and raw IP when a socket is created. A PRU\_ATTACH request is issued by the socket system call. In the case of UDP, we'll see in [Figure 23.33](#) that the resulting call is

```
struct socket *so;
int error;

error = in_pcbaalloc(so, &udb);
```

Figure 22.6 shows the in\_pcalloc function.

## Figure 22.6. in\_pcalloc function: allocate an Internet PCB.

```
36 int  
37 in_pcalloc(so, head)  
38 struct socket *so;  
39 struct inpcb *head;  
40 {  
41     struct inpcb *inp;  
42     MALLOC(inp, struct inpcb *, sizeof(*inp), M_PCB, M_WAITOK);  
43     if (inp == NULL)  
44         return (ENOBUFS);  
45     bzero((caddr_t) inp, sizeof(*inp));  
46     inp->inp_head = head;  
47     inp->inp_socket = so;  
48     insque(inp, head);  
49     so->so_pcb = (caddr_t) inp;  
50     return (0);  
51 }
```

## Allocate PCB and initialize to zero

36-45

in\_pcalloc calls the kernel's memory allocator using the macro MALLOC. Since these PCBs are always allocated as the result of a system call, it is OK to wait for one.

Net/2 and earlier Berkeley releases stored both Internet PCBs and TCP PCBs

in mbufs. Their sizes were 80 and 108 bytes, respectively. With the Net/3 release, the sizes went to 84 and 140 bytes, so TCP control blocks no longer fit into an mbuf. Net/3 uses the kernel's memory allocator instead of mbufs for both types of control blocks.

Careful readers may note that the example in [Figure 2.6](#) shows 17 mbufs allocated for PCBs, yet we just said that Net/3 no longer uses mbufs for Internet PCBs or TCP PCBs. Net/3 does, however, use mbufs for Unix domain PCBs, and that is what this counter refers to. The mbuf statistics output by netstat are for all mbufs in the kernel across all protocol suites, not just the Internet protocols.

bzero sets the PCB to 0. This is important because the IP addresses and port numbers in the PCB must be initialized to 0.

## Link structures together

46-49

The `inp_head` member points to the head of the protocol's PCB list (either `udb` or `tcb`), the `inp_socket` member points to the socket structure, the new PCB is added to the protocol's doubly linked list (`insque`), and the socket structure points to the PCB. The `insque` function puts the new PCB at the head of the protocol's list.

An Internet PCB is deallocated when a `PRU_DETACH` request is issued. This happens when the socket is closed. The function `in_pcbodetach`, shown in [Figure 22.7](#), is eventually called.

## Figure 22.7. `in_pcbodetach` function: deallocate an Internet PCB.

```
252 int                                         in_pcbo.c
253 in_pcbodetach(inp)
254 struct inpcb *inp;
255 {
256     struct socket *so = inp->inp_socket;
257
258     so->so_pcb = 0;
259     soffree(so);
260     if (inp->inp_options)
261         (void) m_free(inp->inp_options);
262     if (inp->inp_route.ro_rt)
263         rtfree(inp->inp_route.ro_rt);
264     ip_freemoptions(inp->inp_moptions);
265     remque(inp);
266 }                                         in_pcbo.c
```

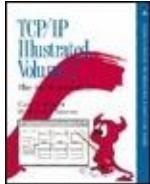
## 252-263

The PCB pointer in the socket structure is set to 0 and that structure is released by sofree. If an mbuf with IP options was allocated for this PCB, it is released by m\_free. If a route is held by this PCB, it is released by rtfree. Any multicast options are also released by ip\_freemoptions.

## 264-265

The PCB is removed from the protocol's doubly linked list by remque and the memory used by the PCB is returned to the kernel.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.5 Binding, Connecting, and Demultiplexing

Before examining the kernel functions that bind sockets, connect sockets, and demultiplex incoming datagrams, we describe the rules imposed by the kernel on these actions.

#### Binding of Local IP Address and Port Number

Figure 22.8 shows the six different combinations of a local IP address and local port number that a process can specify in a call to bind.

## Figure 22.8. Combination of local IP address and local port number for bind.

Local IP address	Local port	Description
unicast or broadcast multicast *	nonzero	one local interface, specific port
	nonzero	one local multicast group, specific port
	nonzero	any local interface or multicast group, specific port
unicast or broadcast multicast *	0	one local interface, kernel chooses port
	0	one multicast group, kernel chooses port
	0	any local interface, kernel chooses port

The first three lines are typical for servers they bind a specific port, termed the server's *well-known port*, whose value is known by the client. The last three lines are typical for clients they don't care what the local port, termed an *ephemeral port*, is, as long as it is unique on the client host.

Most servers and most clients specify the wildcard IP address in the call to bind. This is indicated in Figure 22.8 by the notation \* on lines 3 and 6.

If a server binds a specific IP address to a socket (i.e., not the wildcard address), then only IP datagrams arriving with that specific IP address as the destination IP address be it unicast, broadcast, or

multicast are delivered to the process. Naturally, when the process binds a specific unicast or broadcast IP address to a socket, the kernel verifies that the IP address corresponds to a local interface.

It is rare, though possible, for a client to bind a specific IP address (lines 4 and 5 in [Figure 22.8](#)). Normally a client binds the wildcard IP address (the final line in [Figure 22.8](#)), which lets the kernel choose the outgoing interface based on the route chosen to reach the server.

What we don't show in [Figure 22.8](#) is what happens if the client tries to bind a local port that is already in use with another socket. By default a process cannot bind a port number if that port is already in use. The error EADDRINUSE (address already in use) is returned if this occurs. The definition of *in use* is simply whether a PCB exists with that port as its local port. This notion of "in use" is relative to a given protocol: TCP or UDP, since TCP port numbers are independent of UDP port numbers.

Net/3 allows a process to change this default behavior by specifying one of following two socket options:

**SO\_REUSEADDR** Allows the process to bind a port number that is already in use, but the IP address being bound (including the wildcard) must not already be bound to that same port.

For example, if an attached interface has the IP address 140.252.1.29 then one socket can be bound to 140.252.1.29, port 5555; another socket can be bound to 127.0.0.1, port 5555; and another socket can be bound to the wildcard IP address, port 5555. The call to bind for the second and third cases must be preceded by a call to setsockopt,

setting the `so_reuseaddr` option.

`SO_REUSEPORT` Allows a process to reuse both the IP address and port number, but *each* binding of the IP address and port number, including the first, must specify this socket option. With `SO_REUSEADDR`, the first binding of the port number need not specify the socket option.

For example, if an attached interface has the IP address 140.252.1.29 and a socket is bound to 140.252.1.29, port 6666 specifying the `SO_REUSEPORT` socket option, then another socket can also specify this same socket option

and bind 140.252.1.29,  
port 6666.

Later in this section we describe what happens in this final example when an IP datagram arrives with a destination address of 140.252.1.29 and a destination port of 6666, since two sockets are bound to that end point.

The SO\_REUSEPORT option is new with Net/3 and was introduced with the support for multicasting in 4.4BSD. Before this release it was never possible for two sockets to be bound to the same IP address and same port number.

Unfortunately the so\_REUSEPORT option was not part of the original Stanford multicast sources and is therefore not widely supported. Other systems that support multicasting, such as Solaris 2.x, let a process specify SO\_REUSEADDR to specify that it is OK to bind multiple sockets to the same IP address and same port number.

## Connecting a UDP Socket

We normally associate the connect system call with TCP clients, but it is also possible for a UDP client or a UDP server to call connect and specify the foreign IP address and foreign port number for the socket. This restricts the socket to exchanging UDP datagrams with that one particular peer.

There is a side effect when a UDP socket is connected: the local IP address, if not already specified by a call to bind, is automatically set by connect. It is set to the local interface address chosen by IP routing to reach the specified peer.

[Figure 22.9](#) shows the three different states of a UDP socket along with the pseudocode of the function calls to end up in that state.

**Figure 22.9. Specification of local and foreign IP addresses and port numbers for UDP sockets.**

Local socket	Foreign socket	Description
<i>localIP.lport</i>	<i>foreignIP.fport</i>	restricted to one peer: socket(), bind(*, <i>lport</i> ), connect ( <i>foreignIP</i> , <i>fport</i> ) socket(), bind( <i>localIP</i> , <i>lport</i> ), connect ( <i>foreignIP</i> , <i>fport</i> )
<i>localIP.lport</i>	*.*	restricted to datagrams arriving on one local interface: <i>localIP</i> socket(), bind ( <i>localIP</i> , <i>lport</i> )
*. <i>lport</i>	*.*	receives all datagrams sent to <i>lport</i> : socket(), bind (*, <i>lport</i> )

The first of the three states is called a *connected UDP socket* and the next two states are called *unconnected UDP sockets*. The difference between the two unconnected sockets is that the first has a fully specified local address and the second has a wildcarded local IP address.

## Demultiplexing of Received IP Datagrams by TCP

[Figure 22.10](#) shows the state of three Telnet server sockets on the host sun. The first two sockets are in the LISTEN state, waiting for incoming connection requests, and the third is connected to a client at port 1500 on the host with an IP address of 140.252.1.11. The first listening socket will handle connection requests that arrive on the 140.252.1.29 interface and the second listening socket will handle all other interfaces (since its local IP address

is the wildcard).

**Figure 22.10. Three TCP sockets with a local port of 23.**

Local address	Local port	Foreign address	Foreign port	TCP state
140.252.1.29	23	*	*	LISTEN
*	23	*	*	LISTEN
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED

We show both of the listening sockets with unspecified foreign IP addresses and port numbers because the sockets API doesn't allow a TCP server to restrict either of these values. A TCP server must accept the client's connection and is then told of the client's IP address and port number after the connection establishment is complete (i.e., when TCP's three-way handshake is complete). Only then can the server close the connection if it doesn't like the client's IP address and port number. This isn't a required TCP feature, it is just the way the sockets API has always worked.

When TCP receives a segment with a destination port of 23 it searches through

its list of Internet PCBs looking for a match by calling `in_pcblockup`. When we examine this function shortly we'll see that it has a preference for the smallest number of *wildcard matches*. To determine the number of wildcard matches we consider only the local and foreign IP addresses. We do not consider the foreign port number. The local port number must match, or we don't even consider the PCB. The number of wildcard matches can be 0, 1 (local IP address or foreign IP address), or 2 (both local and foreign IP addresses).

For example, assume the incoming segment is from 140.252.1.11, port 1500, destined for 140.252.1.29, port 23. [Figure 22.11](#) shows the number of wildcard matches for the three sockets from [Figure 22.10](#).

**Figure 22.11. Incoming segment from {140.252.1.11,1500} to {140.252.1.29, 23}.**

Local address	Local port	Foreign address	Foreign port	TCP state	#wildcard matches
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	0

The first socket matches these four values, but with one wildcard match (the foreign IP address). The second socket also matches the incoming segment, but with two wildcard matches (the local and foreign IP addresses). The third socket is a complete match with no wildcards. Net/3 uses the third socket, the one with the smallest number of wildcard matches.

Continuing this example, assume the incoming segment is from 140.252.1.11, port 1501, destined for 140.252.1.29, port 23. [Figure 22.12](#) shows the number of wildcard matches.

**Figure 22.12. Incoming segment from {140.252.1.11, 1501} to {140.252.1.29, 23}.**

Local address	Local port	Foreign address	Foreign port	TCP state	#wildcard matches
140.252.1.29	23	*	*	LISTEN	1
*	23	*	*	LISTEN	2
140.252.1.29	23	140.252.1.11	1500	ESTABLISHED	

The first socket matches with one wildcard match; the second socket matches with two wildcard matches; and the third socket doesn't match at all, since the

foreign port numbers are unequal. (The foreign port numbers are compared only if the foreign IP address in the PCB is not a wildcard.) The first socket is chosen.

In these two examples we never said what type of TCP segment arrived: we assume that the segment in [Figure 22.11](#) contains data or an acknowledgment for an established connection since it is delivered to an established socket. We also assume that the segment in [Figure 22.12](#) is an incoming connection request (a SYN) since it is delivered to a listening socket. But the demultiplexing code in `in_pcblockup` doesn't care. If the TCP segment is the wrong type for the socket that it is delivered to, we'll see later how TCP handles this. For now the important fact is that the demultiplexing code only compares the source and destination socket pair from the IP datagram against the values in the PCB.

## Demultiplexing of Received IP Datagrams by UDP

The delivery of UDP datagrams is more complicated than the TCP example we just examined, since UDP datagrams can be sent to a broadcast or multicast address. Since Net/3 (and most systems with multicast support) allow multiple sockets to have identical local IP addresses and ports, how are multiple recipients handled? The Net/3 rules are:

- 1. An incoming UDP datagram destined for either a broadcast IP address or a multicast IP address is delivered to *all* matching sockets. There is no concept of a "best" match here (i.e., the one with the smallest number of wildcard matches).**
  - An incoming UDP datagram destined for a unicast IP address is delivered only to *one* matching socket, the one with the smallest number of wildcard matches. If there are multiple sockets with the same "smallest" number of wildcard matches, which socket receives the incoming datagram is implementation-dependent.

[Figure 22.13](#) shows four UDP sockets that we'll use for some examples. Having four UDP sockets with the same local port number requires using either SO\_REUSEADDR or SO\_REUSEPORT. The first two sockets have been connected to a foreign IP address and port number, and the last two are unconnected.

### **Figure 22.13. Four UDP sockets with a local port of 577.**

Local address	Local port	Foreign address	Foreign port	Comment
140.252.1.29	577	140.252.1.11	1500	connected, local IP = unicast
140.252.13.63	577	140.252.13.35	1500	connected, local IP = broadcast
140.252.13.63	577	*	*	unconnected, local IP = broadcast
*	577	*	*	unconnected, local IP = wildcard

Consider an incoming UDP datagram destined for 140.252.13.63 (the broadcast address on the 140.252.13 subnet), port 577, from 140.252.13.34, port 1500. [Figure 22.14](#) shows that it is delivered to the third and fourth sockets.

### **Figure 22.14. Received datagram from {140.252.13.34, 1500} to {140.252.13.63, 577}.**

Local address	Local port	Foreign address	Foreign port	Delivered?
140.252.1.29	577	140.252.1.11	1500	no, local and foreign IP mismatch
140.252.13.63	577	140.252.13.35	1500	no, foreign IP mismatch
140.252.13.63	577	*	*	yes
*	577	*	*	yes

The broadcast datagram is not delivered to the first socket because the local IP address doesn't match the destination IP address and the foreign IP address doesn't match the source IP address. It isn't delivered to the second socket because the foreign IP address doesn't match the source IP address.

As the next example, consider an incoming UDP datagram destined for 140.252.1.29 (a unicast address), port 577, from 140.252.1.11, port 1500. [Figure 22.15](#) shows to which sockets the datagram is delivered.

**Figure 22.15. Received datagram from {140.252.1.11, 1500} to {140.252.1.29, 577}.**

Local address	Local port	Foreign address	Foreign port	Delivered?
140.252.1.29	577	140.252.1.11	1500	yes, 0 wildcard matches
140.252.13.63	577	140.252.13.35	1500	no, local and foreign IP mismatch
140.252.13.63	577	*	*	no, local IP mismatch
*	577	*	*	no, 2 wildcard matches

The datagram matches the first socket with no wildcard matches and also matches the fourth socket with two wildcard matches. It is delivered to the first socket, the best match.

---

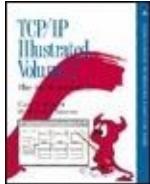
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.6 in\_pcblockup Function

The function `in_pcblockup` serves four different purposes.

- 1. When either TCP or UDP receives an IP datagram, `in_pcblockup` scans the protocol's list of Internet PCBs looking for a matching PCB to receive the datagram. This is transport layer demultiplexing of a received datagram.**

- When a process executes the bind system call, to assign a local IP address and local port number to a socket, `in_pcbbind` is called by the protocol to

verify that the requested local address pair is not already in use.

- When a process executes the bind system call, requesting an ephemeral port be assigned to its socket, the kernel picks an ephemeral port and calls `in_pcbbind` to check if the port is in use. If it is in use, the next ephemeral port number is tried, and so on, until an unused port is located.
- When a process executes the connect system call, either explicitly or implicitly, `in_pcbbind` verifies that the requested socket pair is unique. (An implicit call to connect happens when a UDP datagram is sent on an unconnected socket. We'll see this scenario in [Chapter 23](#).)

In cases 2, 3, and 4 `in_pcbbind` calls `in_pcblayout`. Two options confuse the logic of the function. First, a process can specify either the `SO_REUSEADDR` or `SO_REUSEPORT` socket option to say that a duplicate local address is OK.

Second, sometimes a wildcard match is OK (e.g., an incoming UDP datagram can

match a PCB that has a wildcard for its local IP address, meaning that the socket will accept UDP datagrams that arrive on any local interface), while other times a wildcard match is forbidden (e.g., when connecting to a foreign IP address and port number).

In the original Stanford IP multicast code appears the comment that "The logic of `in_pcblockup` is rather opaque and there is not a single comment, ..." The adjective *opaque* is an understatement.

The publicly available IP multicast code available for BSD/386, which is derived from the port to 4.4BSD done by Craig Leres, fixed the overloaded semantics of this function by using `in_pcblockup` only for case 1 above. Cases 2 and 4 are handled by a new function named `in_pcblockconflict`, and case 3 is handled by a new function named `in_uniqueport`. Dividing the original functionality into separate functions is much clearer, but in the Net/3 release, which we're describing in this text, the logic is still

combined into the single function  
in\_pcblockup.

[Figure 22.16](#) shows the in\_pcblockup  
function.

**Figure 22.16. in\_pcblockup function:  
search all the PCBs for a match.**

---

```

405 struct inpcb *
406 in_pcblkup(head, faddr, fport_arg, laddr, lport_arg, flags)
407 struct inpcb *head;
408 struct in_addr faddr, laddr;
409 u_int    fport_arg, lport_arg;
410 int     flags;
411 {
412     struct inpcb *inp, *match = 0;
413     int      matchwild = 3, wildcard;
414     u_short fport = fport_arg, lport = lport_arg;
415
416     for (inp = head->inp_next; inp != head; inp = inp->inp_next) {
417         if (inp->inp_lport != lport)
418             continue;           /* ignore if local ports are unequal */
419
420         wildcard = 0;
421
422         if (inp->inp_laddr.s_addr != INADDR_ANY) {
423             if (laddr.s_addr == INADDR_ANY)
424                 wildcard++;
425             else if (inp->inp_laddr.s_addr != laddr.s_addr)
426                 continue;
427             } else {
428                 if (laddr.s_addr != INADDR_ANY)
429                     wildcard++;
430             }
431
432         if (inp->inp_faddr.s_addr != INADDR_ANY) {
433             if (faddr.s_addr == INADDR_ANY)
434                 wildcard++;
435             else if (inp->inp_faddr.s_addr != faddr.s_addr ||
436                      inp->inp_fport != fport)
437                 continue;
438             } else {
439                 if (faddr.s_addr != INADDR_ANY)
440                     wildcard++;
441             }
442
443         if (wildcard && (flags & INPLOOKUP_WILDCARD) == 0)
444             continue;           /* wildcard match not allowed */
445
446         if (wildcard < matchwild) {
447             match = inp;
448             matchwild = wildcard;
449             if (matchwild == 0)
450                 break;           /* exact match, all done */
451         }
452     }
453     return (match);
454 }
```

---

in\_pcblk.c

The function starts at the head of the protocol's PCB list and potentially goes through every PCB on the list. The variable `match` remembers the pointer to the entry with the best match so far, and `matchwild` remembers the number of wildcards in that match. The latter is initialized to 3,

which is a value greater than the maximum number of wildcard matches that can be encountered. (Any value greater than 2 would work.) Each time around the loop, the variable wildcard starts at 0 and counts the number of wildcard matches for each PCB.

## Compare local port number

416-417

The first comparison is the local port number. If the PCB's local port doesn't match the lport argument, the PCB is ignored.

## Compare local address

419-427

in\_pcblockup compares the local address in the PCB with the laddr argument. If one is a wildcard and the other is not a wildcard, the wildcard counter is incremented. If both are not wildcards, then they must be the same, or this PCB is

ignored. If both are wildcards, nothing changes: they can't be compared and the wildcard counter isn't incremented. [Figure 22.17](#) summarizes the four different conditions.

**Figure 22.17. Four scenarios for the local IP address comparison done by `in_pcblayout`.**

PCB local IP	laddr argument	Description
not *	*	wildcard++
not *	not *	compare IP addresses, skip PCB if not equal
*	*	can't compare
*	not *	wildcard++

## Compare foreign address and foreign port number

428-437

These lines perform the same test that we just described, but using the foreign addresses instead of the local addresses. Also, if both foreign addresses are not wildcards then not only must the two IP addresses be equal, but the two foreign

ports must also be equal. [Figure 22.18](#) summarizes the foreign IP comparisons.

### **Figure 22.18. Four scenarios for the foreign IP address comparison done by `in_pcblockup`.**

PCB foreign IP	faddr argument	Description
not *	*	wildcard++
not *	not *	compare IP addresses and ports, skip PCB if not equal
*	*	can't compare
*	not *	wildcard++

The additional comparison of the foreign port numbers can be performed for the second line of [Figure 22.18](#) because it is not possible to have a PCB with a nonwildcard foreign address and a foreign port number of 0. This restriction is enforced by connect, which we'll see shortly requires a nonwildcard foreign IP address and a nonzero foreign port. It is possible, however, and common, to have a wildcard local address with a nonzero local port. We saw this in [Figures 22.10](#) and [22.13](#).

### **Check if wildcard match allowed**

438-439

The flags argument can be set to INPLOOKUP\_WILDCARD, which means a match containing wildcards is OK. If a match is found containing wildcards (wildcard is nonzero) and this flag was not specified by the caller, this PCB is ignored. When TCP and UDP call this function to demultiplex an incoming datagram, INPLOOKUP\_WILDCARD is always set, since a wildcard match is OK. (Recall our examples using [Figures 22.10](#) and [22.13](#).) But when this function is called as part of the connect system call, in order to verify that a socket pair is not already in use, the flags argument is set to 0.

**Remember best match, return if exact match found**

440-447

These statements remember the best match found so far. Again, the best match is considered the one with the fewest number of wildcard matches. If a match is

found with one or two wildcards, that match is remembered and the loop continues. But if an exact match is found (wildcard is 0), the loop terminates, and a pointer to the PCB with that exact match is returned.

## ExampleDemultiplexing of Received TCP Segment

[Figure 22.19](#) is from the TCP example we discussed with [Figure 22.11](#). Assume `in_pcblockup` is demultiplexing a received datagram from 140.252.1.11, port 1500, destined for 140.252.1.29, port 23. Also assume that the order of the PCBs is the order of the rows in the figure. `laddr` is the destination IP address, `lport` is the destination TCP port, `faddr` is the source IP address, and `fport` is the source TCP port.

**Figure 22.19. laddr = 140.252.1.29, lport = 23, faddr = 140.252.1.11, fport = 1500.**

PCB values				wildcard
Local address	Local port	Foreign address	Foreign port	
140.252.1.29	23	*	*	1
*	23	*	*	2
140.252.1.29	23	140.252.1.11	1500	0

When the first row is compared to the incoming segment, wildcard is 1 (the foreign IP address), flags is set to INPLOOKUP\_WILDCARD, so match is set to point to this PCB and matchwild is set to 1. The loop continues since an exact match has not been found yet. The next time around the loop, wildcard is 2 (the local and foreign IP addresses) and since this is greater than matchwild, the entry is not remembered, and the loop continues. The next time around the loop, wildcard is 0, which is less than matchwild (1), so this entry is remembered in match. The loop also terminates since an exact match has been found and the pointer to this PCB is returned to the caller.

If in\_pcblkup were used by TCP and UDP only to demultiplex incoming datagrams, it could be simplified. First, there's no need to check whether the faddr or laddr arguments are wildcards, since these are the source and destination IP addresses

from the received datagram. Also the flags argument could be removed, along with its corresponding test, since wildcard matches are always OK.

This section has covered the mechanics of the `in_pcblockup` function. We'll return to this function and discuss its meaning after seeing how it is called from the `in_pcbbind` and `in_pcconnect` functions.



## Chapter 22. Protocol Control Blocks

---

### 22.7 in\_pcbbind Function

The next function, `in_pcbbind`, binds a local address to a socket. It is called from five functions:

#### 1. **from bind for a TCP socket (normally to a well-known port);**

- from bind for a UDP socket (either to bind a socket to a well-known port or to bind an ephemeral port to a client's socket);
- from connect for a TCP socket, if the socket has a nonzero port (this is typical for TCP clients);
- from listen for a TCP socket, if the socket has a nonzero port (this is rare, since listen is called by a TCP server to bind to a well-known port, not an ephemeral port); and
- from `in_pcbbconnect` ([Section 22.8](#)), if the local address and port have not been set (typical for a call to `connect`).

to sendto for an unconnected UDP socket).

In cases 3, 4, and 5, an ephemeral port number and local IP address is not changed (in case it is already bound).

We call cases 1 and 2 *explicit binds* and cases 3–5 *implicit binds*. Note that although it is normal in case 2 for a server to bind to a local IP address and then register their ephemeral port with an RPC program, it is also possible to map between the server's RPC program number and the Sun port mapper described in Section 29.4.

We'll show the `in_pcbbind` function in three sections.

## Figure 22.20. `in_pcbbind` function: bind a local port

```
in_pcbbind.c
52 int
53 in_pcbbind(inp, nam)
54 struct inpcb *inp;
55 struct mbuf *nam;
56 {
57     struct socket *so = inp->inp_socket;
58     struct inpcb *head = inp->inp_head;
59     struct sockaddr_in *sin;
60     struct proc *p = curproc; /* XXX */
61     u_short lport = 0;
62     int      wild = 0, reuseport = (so->so_options & SO_REUSEPORT);
63     int      error;

64     if (in_ifaddr == 0)
65         return (EADDRNOTAVAIL);
66     if (inp->inp_lport || inp->inp_laddr.s_addr != INADDR_ANY)
67         return (EINVAL);

68     if (((so->so_options & (SO_REUSEADDR | SO_REUSEPORT)) == 0 &&
69          ((so->so_proto->pr_flags & PR_CONNREQUIRED) == 0 || 
70           (so->so_options & SO_ACCEPTCONN) == 0))
71         wild = INPLOOKUP_WILDCARD;
```

64-67

The first two tests verify that at least one interface has an address and that the socket is not already bound to it.

68-71

This if statement is confusing. The net result sets the `INPLOOKUP_WILDCARD` flag if neither `SO_REUSEADDR` nor `SO_REUSEPORT` is set.

The second test is true for UDP sockets since `PRU_LISTEN` is valid for connectionless sockets and true for connection-oriented servers.

The third test is where the confusion lies [Torek et al., 2001]. `SO_ACCEPTCONN` is set only by the listen system call and is therefore valid only for a connection-oriented server. In the code, the process calls `socket`, `bind`, and then `listen`. Therefore, when the process calls `listen`, `SO_ACCEPTCONN` is cleared. Even if the process calls `listen` without calling `bind`, TCP's `PRU_LISTEN` request binds an ephemeral port to the socket *before* the `SO_ACCEPTCONN` flag. This means the third test in the if statement, which checks if `SO_ACCEPTCONN` is not set, is always true. This is equivalent to:

```
if ((so->so_options & (SO_REUSEADDR | SO_REUSEPORT)) == 0) {  
    /* ... */  
    wild = INPLOOKUP_WILDCARD;  
}
```

Since anything logically ORed with 1 is always true:

```
if ((so->so_options & (SO_REUSEADDR |  
    wild = INPLOOKUP_WILDCARD))
```

which is simpler to understand: if either of the options is set, it is left as 1. If neither of the REUSE socket options is set, then wild is set to INPLOOKUP\_WILDCARD. In other words, when the INPLOOKUP\_WILDCARD option is specified, a wildcard match is allowed only if neither of the two options are on.

The next section of the in\_pcbbind, shown in Figure 22.21, discusses the optional nam argument.

72-75

The nam argument is a nonnull pointer only when explicitly specified. For an implicit bind (a side effect of cases 3, 4, and 5 from the beginning of this section), the nam argument is specified, it is an mbuf containing the name. Figure 22.21 shows the four cases for the nam argument.

**Figure 22.21. Four cases for nam argument.**

nam argument:		PCB member gets set to:		Comment
<i>localIP</i>	<i>lport</i>	<i>inp_laddr</i>	<i>inp_lport</i>	
not *	0	<i>localIP</i>	ephemeral port <i>lport</i>	<i>localIP</i> must be local interface subject to in_pcblkup
not *	nonzero	<i>localIP</i>	*	
*	0	*	ephemeral port <i>lport</i>	subject to in_pcblkup
*	nonzero	*	*	

76-83

The test for the correct address family is comm  
the in\_pcblkup function (Figure 22.25) is pe  
be in or both to be out.

**Figure 22.22. in\_pcbbind function: proc**

---

```

72     if (nam) {
73         sin = mtod(nam, struct sockaddr_in *);
74         if (nam->m_len != sizeof(*sin))
75             return (EINVAL);
76 #ifdef notdef
77         /*
78          * We should check the family, but old programs
79          * incorrectly fail to initialize it.
80          */
81         if (sin->sin_family != AF_INET)
82             return (EAFNOSUPPORT);
83 #endif
84         lport = sin->sin_port; /* might be 0 */
85         if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr))) {
86             /*
87              * Treat SO_REUSEADDR as SO_REUSEPORT for multicast;
88              * allow complete duplication of binding if
89              * SO_REUSEPORT is set, or if SO_REUSEADDR is set
90              * and a multicast address is bound on both
91              * new and duplicated sockets.
92              */
93             if (so->so_options & SO_REUSEADDR)
94                 reuseport = SO_REUSEADDR | SO_REUSEPORT;
95             } else if (sin->sin_addr.s_addr != INADDR_ANY) {
96                 sin->sin_port = 0; /* yech... */
97                 if (ifa_ifwithaddr((struct sockaddr *) sin) == 0)
98                     return (EADDRNOTAVAIL);
99             }
100            if (lport) {
101                struct inpcb *t;
102                /* GROSS */
103                if (ntohs(lport) < IPPORT_RESERVED &&
104                    (error = suser(p->p_ucred, &p->p_acflag)))
105                    return (error);
106                t = in_pcblklookup(head, zeroin_addr, 0,
107                                   sin->sin_addr, lport, wild);
108                if (t && (reuseport & t->inp_socket->so_options) == 0)
109                    return (EADDRINUSE);
110                }
111                inp->inp_laddr = sin->sin_addr; /* might be wildcard */
112            }

```

---

*in\_pcblk.c*

85-94

Net/3 tests whether the IP address being bound by the SO\_REUSEADDR option is considered identical.

95-99

Otherwise, if the local address being bound by ifa\_ifwithaddr verifies that the address corresponds.

The comment "yech" is probably because the address structure must be 0 because ifa\_ifwithaddr of the entire structure, not just a comparison

This is one of the few instances where the process address structure before issuing the system call 8 bytes of the socket address structure (sin\_zero). ifa\_ifwithaddr will not find the requested interface and return an error.

## 100-105

The next if statement is executed when the call is, the process wants to bind one particular port (one of the scenarios from [Figure 22.21](#)). If the requested port number (IPPORT\_RESERVED) the process must have sufficient memory to store the address of the Internet protocols, but a Berkeley convention states that any port number less than 1024 is called a *reserved port* and is used, for example, by the well-known ports [Stevens 1990], which in turn is used by the remote users for part of their authentication with their servers.

## 106-109

The function in\_pcblklookup ([Figure 22.16](#)) is the function that is called when a connection already exists with the same local IP address and port number. The first argument is the wildcard IP address (the foreign IP address), the second argument is the port number of 0 (the foreign port number), and the third argument causes in\_pcblklookup to ignore the foreign port number.

foreign port in the PCB only the local IP address sin->sin\_addr and lport, respectively. We mention INPLOOKUP\_WILDCARD only if neither of the R

111

The caller's value for the local IP address is stored as a wildcard address, if that's the value specified by the caller. The local IP address is chosen by the kernel, but not until a later time. This is because the local IP address depends on the foreign IP address.

The final section of in\_pcbbind handles the assignment of a local port when the caller explicitly binds a port of 0, or when it is a null pointer (an implicit bind).

**Figure 22.23. in\_pcbbind function: code fragment**

```
113     if (lport == 0)
114         do {
115             if (head->inp_lport++ < IPPORT_RESERVED ||
116                 head->inp_lport > IPPORT_USERRESERVED)
117                 head->inp_lport = IPPORT_RESERVED;
118             lport = htons(head->inp_lport);
119         } while (!in_pcblockup(head,
120                               zeroIn_addr, 0, inp->inp_laddr, lport, wild));
121     inp->inp_lport = lport;
122     return (0);
123 }
```

113-122

The next ephemeral port number to use for this

maintained in the head of the protocol's PCB lists `inp_next` and `inp_back` pointers in the protocol's element of the `inpcb` structure that is used is tracked. This local port number is maintained in host byte order in all the other PCBs on the list starting at 1024 (`IPPORT_RESERVED`) and get increased (`IPPORT_USERRESERVED`), then cycle back until `in_pcbbind` does not find a match.

## **so\_reuseaddr Examples**

Let's look at some common examples to see the behavior of `in_pcbbind` and the two REUSE socket options.

- 1. A TCP or UDP server normally starts by calling bind with a specific address and its nonzero well-known port. Assume a TCP server that calls bind, specifying a specific address and its nonzero well-known port. Also assume that the server is not already listening on that port. The process does not set the SO\_REUSEADDR option.**

**in\_pcbbind calls in\_pcblkup with INP\_NOREUSE argument. The loop in in\_pcblkup finds no entry for the specified port, so it returns 0.**

- Assume the same scenario as above, but with the `SO_REUSEADDR` option set.

someone tries to start the server a second time

When `in_pcbl lookup` is called it finds the PCB with the wildcard counter is 0, `in_pcbl lookup` returns reuseport is 0, `in_pcbb bind` returns EADDRINUSI

- Assume the same scenario as the previous example made to start the server a second time, the SO\_REUSEADDR option is specified.

Since this socket option is specified, `in_pcbb bind` argument of 0. But the PCB with a local socket returned because wildcard is 0, since `in_pcbl lookup` wildcard addresses ([Figure 22.17](#)). `in_pcbb bind` preventing us from starting two instances of the sockets, regardless of whether we specify SO\_REUSEADDR

- Assume that a Telnet server is already running and we try to start another with a local socket {140.252.13.35, 23}.

Assuming SO\_REUSEADDR is not specified, `in_pcbl lookup` argument of INPLOOKUP\_WILDCARD. When it finds a match with the wildcard counter set to 0.23, the counter wildcard is set to 1. Since a wildcard match is remembered as the best match and a local socket is found, the TCP PCBs are scanned. `in_pcbb bind` returns EADDRINUSI

- This example is the same as the previous one except that we specified the SO\_REUSEADDR socket option for the second socket {140.252.13.35, 23}.

The final argument to `in_pcblockup` is now 0, so the wildcard counter is 1, but since the final flags argument is 0, this entry is remembered as a match. After comparing all the TCP PCBs, we find a null pointer and `in_pcbbind` returns 0.

- Assume the first Telnet server is started with {\*, 23} when we try to start a second server with {127.0.0.1, 23} the same as the previous example, except we're changing the order this time.

The first server is started without a problem, as it has already bound port 23. When we start the second server, `in_pcblockup` is `INPLOOKUP_WILDCARD`, assuming the wildcard option is not specified. When the PCB with the entry {\*, 23} is compared, the wildcard counter is set to 1. After all the TCP PCBs are compared, the pointer is still null, causing `in_pcbbind` to return `EADDRINUSE`.

- What if we start two instances of a server, both with the same local address? Assume we start the first Telnet server with {140.252.13.35, 23} and then try to start a second server with {127.0.0.1, 23}, without specifying `SO_REUSEADDR`.

When the second server calls `in_pcbbind`, it calls `in_pcblockup` with the argument of `INPLOOKUP_WILDCARD`. When the PCB with the entry {140.252.13.35, 23} is compared, it is skipped because the addresses are not equal. `in_pcblockup` returns a null pointer.

From this example we see that the SO\_REUSEA option has no effect on nonwildcard IP addresses. Indeed the test of the INPLOOKUP\_WILDCARD in in\_pcblklookup is made only if the value is greater than 0, that is, when either the PCB entry has a wildcard or the address being bound is the wildcard.

- As a final example, assume we try to start two servers, both with the same nonwildcard local IP address.

When the second server is started, in\_pcblklookup finds a matching PCB with the same local socket. This time the SO\_REUSEADDR socket option, because the wildcard comparison. Since in\_pcblklookup returns a nonzero value, the EADDRINUSE.

From these examples we can state the rules about the addresses and the SO\_REUSEADDR socket option. See [Figure 22.24](#). We assume that *localIP1* and *localIP2* are broadcast IP addresses valid on the local host, and *multicastIP* is a multicast group. We also assume that the process has a nonzero port number that is already bound to the local port.

**Figure 22.24. Effect of SO\_REUSEADDR socket option on address.**

Existing PCB	Try to bind	SO_REUSEADDR		Description
		off	on	
<i>localIP1</i>	<i>localIP1</i>	error	error	one server per IP address and port
<i>localIP1</i>	<i>localIP2</i>	OK	OK	one server for each local interface
<i>localIP1</i>	*	error	OK	one server for one interface, other server for remaining interfaces
*	<i>localIP1</i>	error	OK	one server for one interface, other server for remaining interfaces
*	*	error	error	can't duplicate local sockets (same as first example)
<i>localmulticastIP</i>	<i>localmulticastIP</i>	error	OK	multiple multicast recipients

We need to differentiate between a unicast or broadcast address, because we saw that in\_pcbbind considers the same as SO\_REUSEPORT for a multicast address.

## SO\_REUSEPORT Socket Option

The handling of SO\_REUSEPORT in Net/3 changes to allow duplicate local sockets as long as both sockets agree on the words, all the servers must agree to share the same port.

## Chapter 22. Protocol Control Blocks

---

### 22.8 in\_pcbconnect Function

The function `in_pcbconnect` specifies the foreign socket. It is called from four functions:

#### **1. from connect for a TCP socket (required)**

- from connect for a UDP socket (optional for a UDP socket)
- from sendto when a datagram is output on an interface
- from `tcp_input` when a connection request (a SYN) arrives in the LISTEN state (standard for a TCP server).

In all four cases it is common, though not required, for the local values to be unspecified when `in_pcbconnect` is called. Therefore, the local values when they are unspecified.

We'll discuss the `in_pcbconnect` function in four parts:

## Figure 22.25. in\_pcconnect function: validate arguments

```
in_pcconnect.c
130 int
131 in_pcconnect(inp, nam)
132 struct inpcb *inp;
133 struct mbuf *nam;
134 {
135     struct in_ifaddr *ia;
136     struct sockaddr_in *ifaddr;
137     struct sockaddr_in *sin = mtod(nam, struct sockaddr_in *);

138     if (nam->m_len != sizeof(*sin))
139         return (EINVAL);
140     if (sin->sin_family != AF_INET)
141         return (EAFNOSUPPORT);
142     if (sin->sin_port == 0)
143         return (EADDRNOTAVAIL);
144     if (in_ifaddr) {
145         /*
146          * If the destination address is INADDR_ANY,
147          * use the primary local address.
148          * If the supplied address is INADDR_BROADCAST,
149          * and the primary interface supports broadcast,
150          * choose the broadcast address for that interface.
151         */
152 #define satosin(sa)      ((struct sockaddr_in *) (sa))
153 #define sintosa(sin)     ((struct sockaddr *) (sin))
154 #define ifatoia(ifa)     ((struct in_ifaddr *) (ifa))
155     if (sin->sin_addr.s_addr == INADDR_ANY)
156         sin->sin_addr = IA_SIN(in_ifaddr)->sin_addr;
157     else if (sin->sin_addr.s_addr == (u_long) INADDR_BROADCAST &&
158              (in_ifaddr->ia_ifp->if_flags & IFF_BROADCAST))
159         sin->sin_addr = satosin(&in_ifaddr->ia_broadaddr)->sin_addr;
160 }
```

## Validate argument

130-143

The nam argument points to an mbuf containing the destination address and port number. These lines validate the arguments by returning EINVAL if the port number is 0.

## Handle connection to 0.0.0.0 and 255.255.255

144-160

The test of the global in\_ifaddr verifies that an IP address is 0.0.0.0 (INADDR\_ANY), then 0.0. primary IP interface. This means the calling prc foreign IP address is 255.255.255.255 (INADDI supports broadcasting, then 255.255.255.255 i primary interface. This allows a UDP applicatior having to figure out its IP addressit can simply kernel converts this to the appropriate IP addre

The next section of code, [Figure 22.26](#), handles the common scenario for TCP and UDP clients, of this section.

**Figure 22.26. in\_pcbsconnect function**

```

161     if (inp->inp_laddr.s_addr == INADDR_ANY) {
162         struct route *ro;
163
164         ia = (struct in_ifaddr *) 0;
165         /*
166          * If route is known or can be allocated now,
167          * our src addr is taken from the i/f, else punt.
168          */
169         ro = &inp->inp_route;
170         if (ro->ro_rt &&
171             (satosin(&ro->ro_dst)->sin_addr.s_addr !=
172              sin->sin_addr.s_addr ||
173              inp->inp_socket->so_options & SO_DONTROUTE)) {
174             RTFREE(ro->ro_rt);
175             ro->ro_rt = (struct rtentry *) 0;
176         }
177         if ((inp->inp_socket->so_options & SO_DONTROUTE) == 0 && /* XXX */
178             (ro->ro_rt == (struct rtentry *) 0 ||
179              ro->ro_rt->rt_ifp == (struct ifnet *) 0)) {
180             /* No route yet, so try to acquire one */
181             ro->ro_dst.sa_family = AF_INET;
182             ro->ro_dst.sa_len = sizeof(struct sockaddr_in);
183             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
184                 sin->sin_addr;
185             rtalloc(ro);
186         }
187         /*
188          * If we found a route, use the address
189          * corresponding to the outgoing interface
190          * unless it is the loopback (in case a route
191          * to our address on another net goes to loopback).
192          */
193         if (ro->ro_rt && !(ro->ro_rt->rt_ifp->if_flags & IFF_LOOPBACK))
194             ia = ifatoia(ro->ro_rt->rt_ifa);
195         if (ia == 0) {
196             u_short fport = sin->sin_port;
197
198             sin->sin_port = 0;
199             ia = ifatoia(ifa_ifwithdstaddr(sintosa(sin)));
200             if (ia == 0)
201                 ia = ifatoia(ifa_ifwithnet(sintosa(sin)));
202             sin->sin_port = fport;
203             if (ia == 0)
204                 ia = in_ifaddr;
205             if (ia == 0)
206                 return (EADDRNOTAVAIL);
207         }

```

in\_pcbs.c

## Release route if no longer valid

164-175

If a route is held by the PCB but the destination being connected to, or the SO\_DONTROUTE so

To understand why a PCB may have an associat

beginning of this section: `in_pcbconnect` is called on an unconnected socket. Each time a process calls `inet_pcbconnect`, `ip_output`, and `inet_pcbservice` with the same destination IP address, then the first PCB is allocated and it can be used from that point on for datagrams to a different IP address with each connection compared to the saved route and the route released. This test is done in `ip_output`, which seems to be responsible for the route selection.

The `SO_DONTROUTE` socket option tells the kernel not to send the IP datagram to the locally attached interface, but to use the network portion of the destination address.

## Acquire route

176-185

If the `SO_DONTROUTE` socket option is not set, then the kernel tries to acquire one by calling `rtalloc`.

## Determine outgoing interface

186-205

The goal in this section of code is to have `ia_outif` point to the outgoing interface (see [Section 6.5](#)), which contains the IP address of the interface if valid, or if `rtalloc` found a route, and the route information.

corresponding interface is used. Otherwise ifa\_ the foreign IP address is on the other end of a Both of these functions require that the port nu is saved in fport across the calls. If this fails, th no interfaces are configured (in\_ifaddr is zero),

[Figure 22.27](#) shows the next section of in\_pcbs is a multicast address.

## Figure 22.27. in\_pcbs function: d

```
206      /*
207       * If the destination address is multicast and an outgoing
208       * interface has been set as a multicast option, use the
209       * address of that interface as our source address.
210     */
211     if (IN_MULTICAST(ntohl(sin->sin_addr.s_addr)) &&
212         inp->inp_moptions != NULL) {
213         struct ip_moptions *imo;
214         struct ifnet *ifp;
215
216         imo = inp->inp_moptions;
217         if (imo->imo_multicast_ifp != NULL) {
218             ifp = imo->imo_multicast_ifp;
219             for (ia = in_ifaddr; ia; ia = ia->ia_next)
220                 if (ia->ia_ifp == ifp)
221                     break;
222             if (ia == 0)
223                 return (EADDRNOTAVAIL);
224         }
225         ifaddr = (struct sockaddr_in *) &ia->ia_addr;
226     }
```

206-223

If the destination address is a multicast address, the interface to use for multicast packets (using the address of that interface is used as the local address) is the one matching the interface that was specified.

that interface is no longer up.

224-225

The code that started at the beginning of Figure 22.27 is now complete. The pointer to the sockaddr\_in structure in ifaddr is copied into inp\_laddr.

The final section of in\_pcbl lookup is shown in Figure 22.28.

**Figure 22.28. in\_pcconnect function**

```
227     if (in_pcbl lookup(inp->inp_head,
228                         sin->sin_addr,
229                         sin->sin_port,
230                         inp->inp_laddr.s_addr ? inp->inp_laddr : ifaddr->sin_addr,
231                         inp->inp_lport,
232                         0))
233         return (EADDRINUSE);

234     if (inp->inp_laddr.s_addr == INADDR_ANY) {
235         if (inp->inp_lport == 0)
236             (void) in_pcbbind(inp, (struct mbuf *) 0);
237         inp->inp_laddr = ifaddr->sin_addr;
238     }
239     inp->inp_faddr = sin->sin_addr;
240     inp->inp_fport = sin->sin_port;
241     return (0);
242 }
```

in\_pcbl.c

## Verify that socket pair is unique

227-233

in\_pcbl lookup verifies that the socket pair is unique by comparing the values specified as arguments to in\_pcconnect with the values in the

was already bound to the socket or the value is described. The local port can be 0, which is typical in this section of code an ephemeral port is chosen.

This test prevents two TCP connections to the same local address and local port. For example, start a TCP server on the host sun and then try to establish another connection to the same local port (8888, specified with the -b option) causing connect to return the error EADDRINUSE. (See Volume 1.)

```
bsdi $ sock -b 8888 sun echo &
bsdi $ sock -A -b 8888 sun echo
connect () error: Address already in use
```

We specify the -A option to set the SO\_REUSEADDR option but the connect cannot succeed. This is a contradiction since we bind the local port (8888) to both sockets. In the normal case, if we connect to the echo server on the host sun, the local port will be chosen by the kernel via in\_pcblklookup from [Figure 22.28](#).

This test also prevents two UDP sockets from binding to the same local port. This test does not prevent datagrams to the same foreign address from being sent, since a UDP socket is only temporarily bound during a connect system call.

## Implicit bind and assignment of ephemeral port

234-238

If the local address is still wildcarded for the socket, then there is an implicit bind: cases 3, 4, and 5 from the beginning of the function depend on whether the local port has been bound yet, and if so, what port it is. The order of the call to in\_pcbbind is important here since in\_pcbbind fails if the local address is not wildcarded.

## Store foreign address and foreign port in PCB

239-240

The final step of this function sets the foreign IP address and port. We are guaranteed, on successful return from this function, that both the local and foreign addresses and ports are filled in with specific values.

## IP Source Address Versus Outgoing Interface

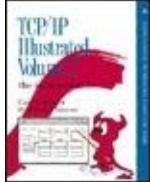
There is a subtle difference between the source IP address and the outgoing interface used to send the datagram.

The PCB member inp\_laddr is used by TCP and UDP to determine the source IP address. It can be set by the process to the IP address of any interface. The function ifa\_ifwithaddr in in\_pcbbind verifies the local address and then assigns the local address only if it is a wildcard.

based on the outgoing interface (since the dest

The outgoing interface, however, is also determined by the destination address. On a multihomed host it is possible for the outgoing interface to be different from the one specified in the routing table. This is allowed because (Section 8.4).

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.9 in\_pcbservice Function

A UDP socket is disconnected by `in_pcbservice`. This removes the foreign association by setting the foreign IP address to all 0s (`INADDR_ANY`) and foreign port number to 0.

This is done after a datagram has been sent on an unconnected UDP socket and when `connect` is called on a connected UDP socket. In the first case the sequence of steps when the process calls `sendto` is: UDP calls `in_pcbservice` to connect the socket temporarily to the destination, `udp_output` sends the datagram, and then `in_pcbservice` removes the temporary

connection.

in\_pcbservice is not called when a socket is closed since in\_pcbservice handles the release of the PCB. A disconnect is required only when the PCB needs to be reused for a different foreign address or port number.

[Figure 22.29](#) shows the function in\_pcbservice.

## Figure 22.29. in\_pcbservice function: disconnect from foreign address and port number.

```
243 int  
244 in_pcbservice(inp)  
245 struct inpcb *inp;  
246 {  
247     inp->inp_faddr.s_addr = INADDR_ANY;  
248     inp->inp_fport = 0;  
249     if (inp->inp_socket->so_state & SS_NOFDREF)  
250         in_pcbservice(inp);  
251 }
```

If there is no longer a file table reference for this PCB (SS\_NOFDREF is set) then in\_pcbservice ([Figure 22.7](#)) releases the PCB.

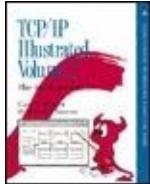
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.10 `in_setsockaddr` and `in_setpeeraddr` Functions

The `getsockname` system call returns the local protocol address of a socket (e.g., the IP address and port number for an Internet socket) and the `getpeername` system call returns the foreign protocol address. Both system calls end up issuing a `PRU_SOCKADDR` request or a `PRU_PEERADDR` request. The protocol then calls either `in_setsockaddr` or `in_setpeeraddr`. We show the first of these in Figure 22.30.

**Figure 22.30. `in_setsockaddr` function:**

## return local address and port number.

```
267 int  
268 in_setsockaddr(inp, nam)  
269 struct inpcb *inp;  
270 struct mbuf *nam;  
271 {  
272     struct sockaddr_in *sin;  
273     nam->m_len = sizeof(*sin);  
274     sin = mtod(nam, struct sockaddr_in *);  
275     bzero((caddr_t) sin, sizeof(*sin));  
276     sin->sin_family = AF_INET;  
277     sin->sin_len = sizeof(*sin);  
278     sin->sin_port = inp->inp_lport;  
279     sin->sin_addr = inp->inp_laddr;  
280 }
```

The argument nam is a pointer to an mbuf that will hold the result: a sockaddr\_in structure that the system call copies back to the process. The code fills in the socket address structure and copies the IP address and port number from the Internet PCB into the sin\_addr and sin\_port members.

Figure 22.31 shows the in\_setpeeraddr function. It is nearly identical to Figure 22.30, but copies the foreign IP address and port number from the PCB.

## Figure 22.31. in\_setpeeraddr function: return foreign address and port number.

```
281 int
282 in_setpeeraddr(inp, nam)
283 struct inpcb *inp;
284 struct mbuf *nam;
285 {
286     struct sockaddr_in *sin;
287     nam->m_len = sizeof(*sin);
288     sin = mtod(nam, struct sockaddr_in *);
289     bzero((caddr_t) sin, sizeof(*sin));
290     sin->sin_family = AF_INET;
291     sin->sin_len = sizeof(*sin);
292     sin->sin_port = inp->inp_fport;
293     sin->sin_addr = inp->inp_faddr;
294 }
```

---

**Team-Fly** 

[◀ Previous](#)

[Next ▶](#)

[Top](#)

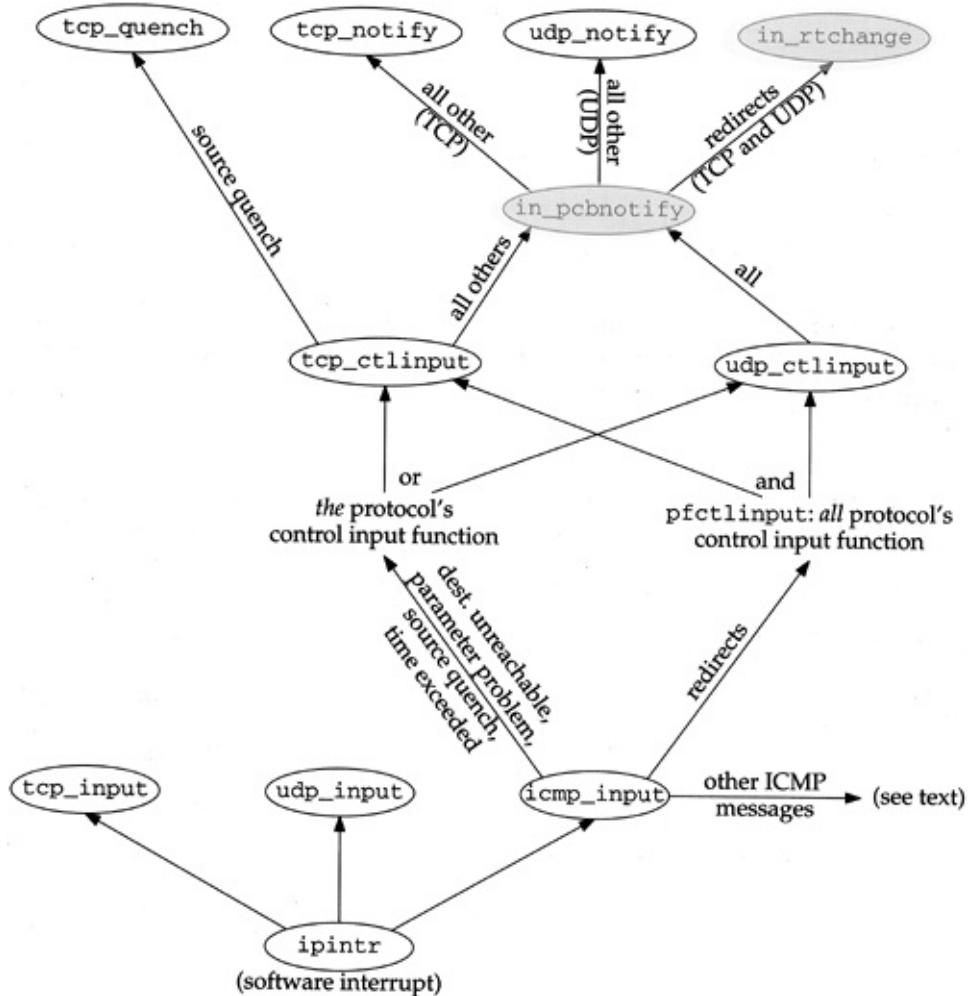
# Chapter 22. Protocol Control Blocks

## 22.11 in\_pcbsnotify, in\_rtchange, and

The function `in_pcbsnotify` is called when an ICMP message arrives, and it performs the appropriate process of the error. The "appropriate process" depends on the protocol. For example, TCP must scan all the PCBs for one of the protocols (TCP or UDP) that have the same source IP address and port number as the error. In the case of IP addresses and port numbers with the values specified in the error message. For example, when an ICMP source quench error is received, TCP must locate the PCB that has the same source IP address and port number as the error message. It then reduces the transmission rate on that connection to prevent the router from discarding the packet.

Before showing the function we must review how the functions called to process an ICMP error. These functions described in this section.

## Figure 22.32. Summary of pro



When an ICMP message is received, `icmp_input` are classified as errors ([Figures 11.1](#) and [11.2](#))

- destination unreachable,
- parameter problem,
- redirect,
- source quench, and

- time exceeded.

Redirects are handled differently from the other errors. Redirects (the queries) are handled as described in [Chapter 21](#).

Each protocol defines its control input function, which is part of its structure ([Section 7.4](#)). The ones for TCP and UDP are called pfctlinput and udp\_ctlinput, and we'll show their code in later chapters. The error message received contains the IP header of the datagram that caused the error (TCP or UDP) is known. If the error is a redirect, the protocol's control input function is called. Redirects are handled specially because they affect the destination, not just the one that caused the redirect. The four errors need only be processed by the protocols in the family (Internet). TCP and UDP have their own family with control input functions.

Redirects are handled specially because they affect the destination, not just the one that caused the redirect. The four errors need only be processed by the protocols in the family (Internet).

The final points we need to make about [Figure 22.32](#) are that it handles quenches differently from the other errors, and it handles in\_pcbsend and in\_pcbnotify: the function in\_rtchange is called when either of these errors occurs.

[Figure 22.33](#) shows the in\_pcbsend and in\_pcbnotify functions. The first argument is the address of tcb and the final argument is the function in\_pcbsend or in\_pcbnotify. For TCP, these two arguments are the function tcp\_notify and the function in\_pcbsend. For UDP, these two arguments are the function udp\_notify and the function in\_pcbnotify.

## Figure 22.33. in\_pcnotify function: pas

```
306 int  
307 in_pcnotify(head, dst, fport_arg, laddr, lport_arg, cmd, notify)  
308 struct inpcb *head;  
309 struct sockaddr *dst;  
310 u_int fport_arg, lport_arg;  
311 struct in_addr laddr;  
312 int cmd;  
313 void (*notify) (struct inpcb *, int);  
314 {  
315     extern u_char inetctllerrmap[];  
316     struct inpcb *inp, *oinp;  
317     struct in_addr faddr;  
318     u_short fport = fport_arg, lport = lport_arg;  
319     int errno;  
320     if ((unsigned) cmd > PRC_NCMDS || dst->sa_family != AF_INET)  
321         return;  
322     faddr = ((struct sockaddr_in *) dst)->sin_addr;  
323     if (faddr.s_addr == INADDR_ANY)  
324         return;  
325     /*  
326      * Redirects go to all references to the destination,  
327      * and use in_rtchange to invalidate the route cache.  
328      * Dead host indications: notify all references to the destination.  
329      * Otherwise, if we have knowledge of the local port and address,  
330      * deliver only to that socket.  
331      */  
332     if (PRC_IS_REDIRECT(cmd) || cmd == PRC_HOSTDEAD) {  
333         fport = 0;  
334         lport = 0;  
335         laddr.s_addr = 0;  
336         if (cmd != PRC_HOSTDEAD)  
337             notify = in_rtchange;  
338     }  
339     errno = inetctllerrmap[cmd];  
340     for (inp = head->inp_next; inp != head;) {  
341         if (inp->inp_faddr.s_addr != faddr.s_addr ||  
342             inp->inp_socket == 0 ||  
343             (lport && inp->inp_lport != lport) ||  
344             (laddr.s_addr && inp->inp_laddr.s_addr != laddr.s_addr) ||  
345             (fport && inp->inp_fport != fport)) {  
346             inp = inp->inp_next;  
347             continue; /* skip this PCB */  
348         }  
349         oinp = inp;  
350         inp = inp->inp_next;  
351         if (notify)  
352             (*notify) (oinp, errno);  
353     }  
354 }
```

in\_pcnotify

## Verify arguments

306-324

The cmd argument and the address family of the address is checked to ensure it is not 0.0.0.0.

## Handle redirects specially

325-338

If the error is a redirect it is handled specially. (An error that was generated by the IMPs. Current (and a historical artifact.) The foreign port, local port comparison in that the for loop that follows won't compare the foreign port to the local port. Instead it will select the PCBs to receive notification based on the IP address of the host that received the redirect. That is the IP address for which our host received the redirect. The function called for a redirect is `in_rtchange` (Figure 22.3) and is specified by the caller.

Figure 22.34. `in_rtchange` function

```
391 void in_rtchange(inp, errno)
392 in_rtchange(inp, errno)
393 struct inpcb *inp;
394 int     errno;
395 {
396     if (inp->inp_route.ro_rt) {
397         rtfree(inp->inp_route.ro_rt);
398         inp->inp_route.ro_rt = 0;
399         /*
400          * A new route can be allocated the next time
401          * output is attempted.
402          */
403     }
404 }
```

339

The global array `inetctllerrmap` maps one of the `PRC_xxx` values from [Figure 11.19](#)) into its corresponding column in [Figure 11.1](#)).

## Call notify function for selected PCBs

340-353

This loop selects the PCBs to be notified. Multiples of 5 are going even after a match is located. The first if any one of the five is true, the PCB is skipped: (1) if the ports are unequal, (2) if the PCB does not have a corresponding entry in the map, (3) if the local addresses are unequal, (4) if the foreign addresses are unequal. The foreign addresses *must* match, while the local addresses are compared only if the corresponding entry is found, the notify function is called.

## in\_rtchange Function

We saw that `in_pcbnotify` calls the function `in_rtchange`. This function is called for all PCBs with a local address that has been redirected. [Figure 22.34](#)

If the PCB holds a route, that route is released and marked as empty. We don't try to update the route's address returned in the redirect. The new route for this PCB is used next, based on the kernel's routing table.

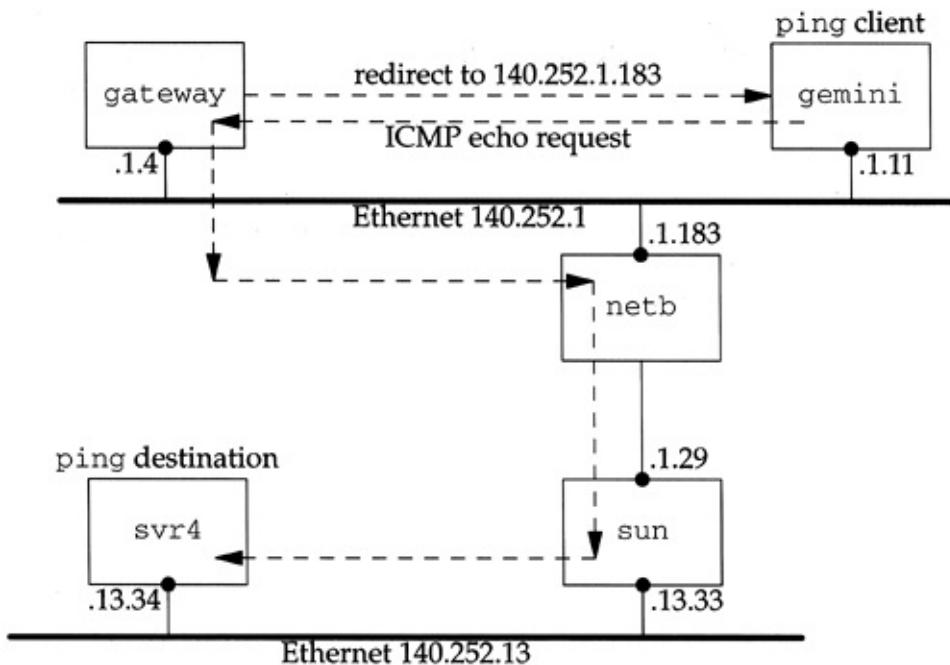
redirect, before pfctlinput is called.

## Redirects and Raw Sockets

Let's examine the interaction of redirects, raw sockets, and the PCB. If we run the Ping program, which uses a raw socket to receive packets, and a redirect is received for the IP address being pinged, Pin will receive the redirected route. We can see this as follows:

We ping the host svr4 on the 140.252.13 network from the host gemini on the 140.252.1 network. The default router for gemini is netb, but the packet will be sent to the router netb instead. [Figure 22.35!](#)

**Figure 22.35. Example of a redirect**



We expect gateway to send a redirect when it receives our ping:

```
gemini $ ping -sv svr4
PING 140.252.13.34: 56 data bytes
ICMP Host redirect from gateway
    to netb (140.252.1.183) for svr4
64 bytes from svr4 (140.252.13.34)

ICMP Host redirect from gateway
    to netb (140.252.1.183) for svr4
64 bytes from svr4 (140.252.13.34)
```

The **-s** option causes an ICMP echo request to be sent, and the **-v** option prints every received ICMP message (including redirects).

Every ICMP echo request elicits a redirect, but the kernel notices the redirect to change the route that it calculated and stored in the PCB, causing the IP address of the gateway (140.252.1.4), which the kernel expects to receive responses from, to be updated so that it routes to the router netb (140.252.1.183) instead. We see the kernel messages in the log output of the kernel on gemini, but they appear to be ignored.

If we terminate the program and start it again,

```
gemini $ ping -sv svr4
PING 140.252.13.34: 56 data bytes
64 bytes from svr4 (140.252.13.34)
```

64 bytes from svr4 (140.252.13.3)

# ICMP Errors and UDP Sockets

One confusing part of the sockets API is that ICMP messages are not passed to the application unless the application has bound to a local socket, restricting the foreign IP address and port number. This is where this limitation is enforced by `in_pcbsnotif`.

Consider an ICMP port unreachable, probably to a socket. The foreign IP address and the foreign in\_pcbsend are the IP address and port number. The process has not issued a connect on the socket of the PCB are both 0, preventing in\_pcbsend from sending this socket. The for loop in Figure 22.33 will skip this socket.

This limitation arises for two reasons. First, if the UDP socket, the only nonzero element in the sc

assumes the process did not call bind.) This is true to demultiplex the incoming ICMP error and pass it to the process. However, there could be multiple processes bound to the same local port, making it ambiguous which process should receive the error. If another process then starts and uses the same local port, the ephemeral ports are assigned in sequential order after cycling around ([Figure 22.23](#)).

The second reason for this limitation is because the process can return an errno value that is inadequate. Consider the following sequence: a process creates an unconnected UDP socket three times in a row, sends datagrams to three different destinations, and then waits for the responses. If one of the responses is a datagram that generates an ICMP port unreachable error, the corresponding error (ECONNREFUSED) is returned. However, the errno value doesn't tell the process which connection generated the error. The kernel has all the information required in the ICMP message, but it doesn't provide a way to return this to the process.

Therefore the design decision was made that if an ICMP error occurs on a UDP socket, that socket must return ECONNREFUSED. This means that the error is returned on that connection, even if the peer generated the error.

There is still a remote possibility of an ICMP error occurring on a UDP socket. One process sends the UDP datagram and terminates before the error is received. Another process receives the error, binds the same local port, and

and foreign port, causing this new process to reprevent this from occurring, given UDP's lack of this with its TIME\_WAIT state.

In our preceding example, one way for the app to use three connected UDP sockets instead of to determine when any one of the three has a read.

Here we have a scenario where the kernel has is inadequate. With most implementations of API (TLI), the reverse is true: the TLI function address, port number, and an error value, but of TCP/IP don't provide a way for ICMP to pass end point.

In an ideal world, in\_pcbsnotify delivers the IC match, even if the only nonwildcard match is the process would include the destination IP address caused the error, allowing the process to determine datagram sent by the process.

## in\_losing Function

The final function dealing with PCBs is in\_losing TCP when its retransmission timer has expired connection ([Figure 25.26](#)).

## Figure 22.36. in\_losing function: inval

```
-----in_pcbo.c
361 int
362 in_losing(inp)
363 struct inpcb *inp;
364 {
365     struct rtentry *rt;
366     struct rt_addrinfo info;
367     if ((rt = inp->inp_route.ro_rt)) {
368         inp->inp_route.ro_rt = 0;
369         bzero((caddr_t) &info, sizeof(info));
370         info.rti_info[RTAX_DST] =
371             (struct sockaddr *) &inp->inp_route.ro_dst;
372         info.rti_info[RTAX_GATEWAY] = rt->rt_gateway;
373         info.rti_info[RTAX_NETMASK] = rt->rt_mask(rt);
374         rt_missmsg(RTM_LOSING, &info, rt->rt_flags, 0);
375         if (rt->rt_flags & RTF_DYNAMIC)
376             (void) rtrequest(RTM_DELETE, rt_key(rt),
377                             rt->rt_gateway, rt->rt_mask(rt), rt->rt_flags,
378                             (struct rtentry **) 0);
379     else
380         /*
381          * A new route can be allocated
382          * the next time output is attempted.
383          */
384         rtfree(rt);
385     }
386 }
```

-----in\_pcbo.c

## Generate routing message

361-374

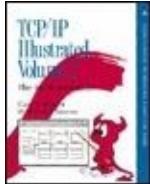
If the PCB holds a route, that route is discarded with information about the cached route that a rt\_missmsg is then called to generate a message RTM\_LOSING, indicating a problem with the route.

## Delete or release route

375-384

If the cached route was generated by a redirect, it is deleted by calling `rtrequest` with a request of `R`. The route is released, causing the next output on the destination hopefully a better route.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.12 Implementation Refinements

Undoubtedly the most time-consuming algorithm we've encountered in this chapter is the linear searching of the PCBs done by `in_pcblklookup`. At the beginning of [Section 22.6](#) we noted four instances when this function is called. We can ignore the calls to bind and connect, as they occur much less frequently than the calls to `in_pcblklookup` from TCP and UDP, to demultiplex every received IP datagram.

In later chapters we'll see that TCP and UDP both try to help this linear search by maintaining a pointer to the last PCB that

the protocol referenced: a one-entry cache. If the local address, local port, foreign address, and foreign port in the cached PCB match the values in the received datagram, the protocol doesn't even call `in_pcblockup`. If the protocol's data fits the packet train model [Jain and Routhier 1986], this simple cache works well. But if the data does not fit this model and, for example, looks like data entry into an on-line transaction processing system, the one-entry cache performs poorly [McKenney and Dove 1992].

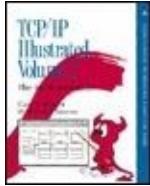
One proposal for a better PCB arrangement is to move a PCB to the front of the PCB list when the PCB is referenced. ([McKenney and Dove 1992] attribute this idea to Jon Crowcroft; [Partridge and Pink 1993] attribute it to Gary Delp.) This movement of the PCB is easy to do since it is a doubly linked list and a pointer to the head of the list is the first argument to `in_pcblockup`.

[McKenney and Dove 1992] compare the original Net/1 implementation (no cache), an enhanced one-entry sendreceive cache,

the move-to-the-front heuristic, and their own algorithm that uses hash chains. They show that maintaining a linear list of PCBs on hash chains provides an order of magnitude improvement over the other algorithms. The only cost for the hash chains is the memory required for the hash chain headers and the computation of the hash function. They also consider adding the move-to-the-front heuristic to their hash-chain algorithm and conclude that it is easier simply to add more hash chains.

Another comparison of the BSD linear search to a hash table search is in [\[Hutchinson and Peterson 1991\]](#). They show that the time required to demultiplex an incoming UDP datagram is constant as the number of sockets increases for a hash table, but with a linear search the time increases as the number of sockets increases.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 22. Protocol Control Blocks

### 22.13 Summary

An Internet PCB is associated with every Internet socket: TCP, UDP, and raw IP. It contains information common to all Internet sockets: local and foreign IP addresses, pointer to a route structure, and so on. All the PCBs for a given protocol are placed on a doubly linked list maintained by that protocol.

In this chapter we've looked at numerous functions that manipulate the PCBs, and three in detail.

- 1. `in_pcblklookup` is called by TCP and UDP to demultiplex every received**

**datagram. It chooses which socket receives the datagram, taking into account wildcard matches.**

**This function is also called by in\_pcbbind to verify that the local address and local process are unique, and by in\_pcconnect to verify that the combination of a local address, local process, foreign address, and foreign process are unique.**

- in\_pcbbind explicitly or implicitly binds a local address and local port to a socket. An explicit bind occurs when the process calls bind, and an implicit bind occurs when a TCP client calls connect without calling bind, or when a UDP process calls sendto or connect without calling bind.
- in\_pcconnect sets the foreign address and foreign process. If the local address has not been set by the process, a route to the foreign address is calculated and the resulting local interface becomes the local address. If the local port has not been set by the process, in\_pcbbind chooses an

ephemeral port for the socket.

[Figure 22.37](#) summarizes the common scenarios for various TCP and UDP applications and the values stored in the PCB for the local address and port and the foreign address and port. We have not yet covered all the actions shown in [Figure 22.37](#) for TCP and UDP processes, but will examine the code in later chapters.

**Figure 22.37. Summary of `in_pcbbind` and `in_pcbconnect`.**

Application	local address: inp_laddr	local port: inp_lport	foreign address: inp_faddr	foreign port: inp_fport
TCP client: <code>connect (foreignIP, fport)</code>	in_pcbsconnect calls rtalloc to allocate route to <i>foreignIP</i> . Local address is local interface.	in_pcbsconnect calls in_pcbbind to choose ephemeral port.	<i>foreignIP</i>	<i>fport</i>
TCP client: <code>bind (localIP, lport)</code> <code>connect (foreignIP, fport)</code>	<i>localIP</i>	<i>lport</i>	<i>foreignIP</i>	<i>fport</i>
TCP client: <code>bind (*, lport)</code> <code>connect (foreignIP, fport)</code>	in_pcbsconnect calls rtalloc to allocate route to <i>foreignIP</i> . Local address is local interface.	<i>lport</i>	<i>foreignIP</i>	<i>fport</i>
TCP client: <code>bind (localIP, 0)</code> <code>connect (foreignIP, fport)</code>	<i>localIP</i>	in_pcbbind chooses ephemeral port.	<i>foreignIP</i>	<i>fport</i>
TCP server: <code>bind (localIP, lport)</code> <code>listen()</code> <code>accept()</code>	<i>localIP</i>	<i>lport</i>	Source address from IP header.	Source port from TCP header.
TCP server: <code>bind (*, lport)</code> <code>listen()</code> <code>accept()</code>	Destination address from IP header.	<i>lport</i>	Source address from IP header.	Source port from TCP header.
UDP client: <code>sendto (foreignIP, fport)</code>	in_pcbsconnect calls rtalloc to allocate route to <i>foreignIP</i> . Local address is local interface. Reset to 0.0.0.0 after datagram sent.	in_pcbsconnect calls in_pcbbind to choose ephemeral port. Not changed on subsequent calls to sendto.	<i>foreignIP</i> . Reset to 0.0.0.0 after datagram sent.	<i>fport</i> . Reset to 0 after datagram sent.
UDP client: <code>connect (foreignIP, fport)</code> <code>write()</code>	in_pcbsconnect calls rtalloc to allocate route to <i>foreignIP</i> . Local address is local interface. Not changed on subsequent calls to write.	in_pcbsconnect calls in_pcbbind to choose ephemeral port. Not changed on subsequent calls to write.	<i>foreignIP</i>	<i>fport</i>

## Exercises

**22.1** What happens in [Figure 22.23](#) when the process asks for an ephemeral port and every ephemeral port is in use?

In Figure 22.10 we showed two Telnet servers with listening sockets: one with a specific local IP address

**22.2** and one with the wildcard for its local IP address. Does your system's Telnet daemon allow you to specify the local IP address, and if so, how?

Assume a socket is bound to the local socket (140.252.1.29, 8888), and this is the only socket using local port 8888. (1) Go through the steps performed by `in_pcbbind` when another socket is bound to (140.252.13.33, 8888), without any socket options. (2) Go through the steps performed when another socket is bound to the wildcard IP address, port 8888, without any socket options. (3) Go through the steps performed when another socket is bound to the wildcard IP address, port 8888, with the `SO_REUSEADDR` socket option.

**22.3**

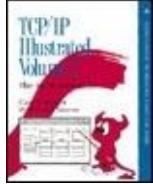
What is the first ephemeral port

**22.4** number allocated by UDP?

When a process calls bind, which  
**22.5** elements in the sockaddr\_in  
structure must be filled in?

What happens if a process tries to  
bind a local broadcast address? What  
**22.6** happens if a process tries to bind the  
limited broadcast address  
(255.255.255.255)?





**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 23. UDP: User Datagram Protocol

[Section 23.1. Introduction](#)

[Section 23.2. Code Introduction](#)

[Section 23.3. UDP protosw Structure](#)

[Section 23.4. UDP Header](#)

[Section 23.5. udp\\_init Function](#)

[Section 23.6. udp\\_output Function](#)

[Section 23.7. udp\\_input Function](#)

[Section 23.8. udp\\_saveopt Function](#)

[Section 23.9. udp\\_ctlinput Function](#)

[Section 23.10. udp\\_usrreq Function](#)

[Section 23.11. udp\\_sysctl Function](#)

[Section 23.12. Implementation  
Refinements](#)

## Section 23.13. Summary

---

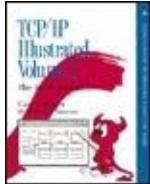
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.1 Introduction

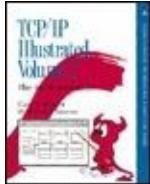
The User Datagram Protocol, or UDP, is a simple, datagram-oriented, transport-layer protocol: each output operation by a process produces exactly one UDP datagram, which causes one IP datagram to be sent.

A process accesses UDP by creating a socket of type `SOCK_DGRAM` in the Internet domain. By default the socket is termed *unconnected*. Each time the process sends a datagram it must specify the destination IP address and port number. Each time a datagram is received for the socket, the process can receive the

source IP address and port number from the datagram.

We mentioned in [Section 22.5](#) that a UDP socket can also be *connected* to one particular IP address and port number. This causes all datagrams written to the socket to go to that destination, and only datagrams arriving from that IP address and port number are passed to the process.

This chapter examines the implementation of UDP.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.2 Code Introduction

There are nine UDP functions in a single C file and various UDP definitions in two headers, as shown in [Figure 23.1](#).

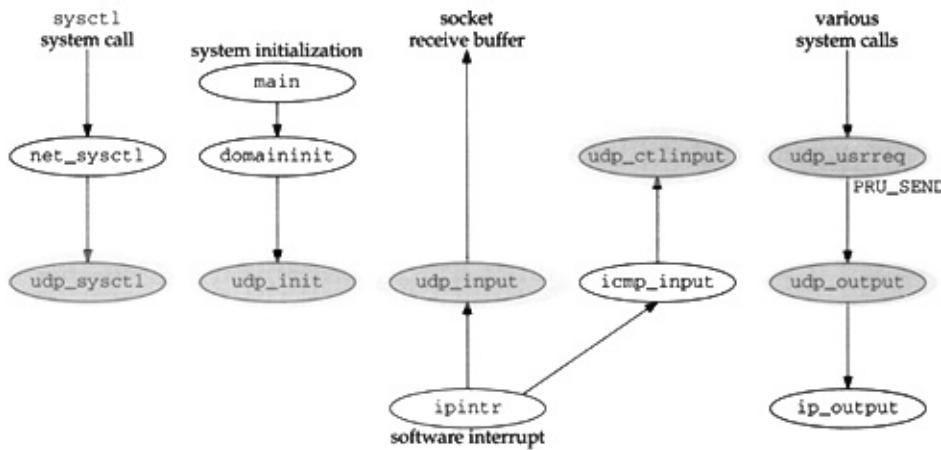
**Figure 23.1. Files discussed in this chapter.**

File	Description
netinet/udp.h	udphdr structure definition
netinet/udp_var.h	other UDP definitions
netinet/udp_usrreq.c	UDP functions

[Figure 23.2](#) shows the relationship of the six main UDP functions to other kernel functions. The shaded ellipses are the six

functions that we cover in this chapter. We also cover three additional UDP functions that are called by some of these six functions.

**Figure 23.2. Relationship of UDP functions to rest of kernel.**



## Global Variables

Seven global variables are introduced in this chapter, which are shown in [Figure 23.3](#).

**Figure 23.3. Global variables introduced in this chapter.**

Variable	Datatype	Description
udb	struct inpcb	head of the UDP PCB list
udp_last_inpcb	struct inpcb *	pointer to PCB for last received datagram: one-behind cache
udpcksum	int	flag for calculating and verifying UDP checksum
udp_in	struct sockaddr_in	holds sender's IP address and port on input
udpstat	struct udpstat	UDP statistics (Figure 23.4)
udp_recvspace	u_long	default size of socket receive buffer, 41,600 bytes
udp_sendspace	u_long	default size of socket send buffer, 9216 bytes

## Statistics

Various UDP statistics are maintained in the global structure `udpstat`, described in [Figure 23.4](#). We'll see where these counters are incremented as we proceed through the code.

**Figure 23.4. UDP statistics maintained in the `udpstat` structure.**

udpstat member	Description	Used by SNMP
udps_badlen	#received datagrams with data length larger than packet	•
udps_badsum	#received datagrams with checksum error	•
udps_fullsock	#received datagrams not delivered because input socket full	•
udps_hdrops	#received datagrams with packet shorter than header	•
udps_ipackets	total #received datagrams	•
udps_noport	#received datagrams with no process on destination port	•
udps_noportbcast	#received broadcast/multicast datagrams with no process on dest. port	•
udps_opackets	total #output datagrams	•
udpps_pcbcachemiss	#received input datagrams missing pcb cache	•

[Figure 23.5](#) shows some sample output of these statistics, from the `netstat -s` command.

## Figure 23.5. Sample UDP statistics.

netstat -s output	udpstat member
18,575,142 datagrams received	udps_ipackets
0 with incomplete header	udps_hdrops
18 with bad data length field	udps_badlen
58 with bad checksum	udps_badsum
84,079 dropped due to no socket	udps_noport
446 broadcast/multicast datagrams dropped due to no socket	udps_noportbcast
5,356 dropped due to full socket buffers	udps_fullsock
18,485,185 delivered	(see text)
18,676,277 datagrams output	udps_opackets

The number of UDP datagrams delivered (the second from last line of output) is the number of datagrams received (udps\_ipackets) minus the six variables that precede it in [Figure 23.5](#).

## SNMP Variables

[Figure 23.6](#) shows the four simple SNMP variables in the UDP group and which counters from the udpstat structure implement that variable.

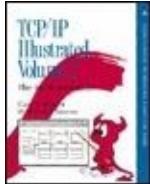
## Figure 23.6. Simple SNMP variables in udp group.

SNMP variable	udpstat member	Description
udpInDatagrams	udps_ipackets	#received datagrams delivered to processes
udpInErrors	udps_hdrops + udps_badsum + udps_badlen	#undeliverable UDP datagrams for reasons other than no application at destination port (e.g., UDP checksum error)
udpNoPorts	udps_noport + udps_noportbroadcast	#received datagrams for which no application process was at the destination port
udpOutDatagrams	udps_opackets	#datagrams sent

Figure 23.7 shows the UDP listener table, named `udpTable`. The values returned by SNMP for this table are taken from a UDP PCB, not the `udpstat` structure.

## Figure 23.7. Variables in UDP listener table: `udpTable`.

UDP listener table, index = < <code>udpLocalAddress</code> >.< <code>udpLocalPort</code> >		
SNMP variable	PCB variable	Description
<code>udpLocalAddress</code>	<code>inp_laddr</code>	local IP address for this listener
<code>udpLocalPort</code>	<code>inp_lport</code>	local port number for this listener



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.3 UDP protosw Structure

Figure 23.8 lists the protocol switch entry for UDP.

**Figure 23.8. The UDP protosw structure.**

Member	inetsw[1]	Description
pr_type	<code>SOCK_DGRAM</code>	UDP provides datagram packet services
pr_domain	<code>&amp;inetdomain</code>	UDP is part of the Internet domain
pr_protocol	<code>IPPROTO_UDP (17)</code>	appears in the <code>ip_p</code> field of the IP header
pr_flags	<code>PR_ATOMIC PR_ADDR</code>	socket layer flags, not used by protocol processing
pr_input	<code>udp_input</code>	receives messages from IP layer
pr_output	<code>0</code>	not used by UDP
pr_ctlinput	<code>udp_ctlinput</code>	control input function for ICMP errors
pr_ctloutput	<code>ip_ctloutput</code>	respond to administrative requests from a process
pr_usrreq	<code>udp_usrreq</code>	respond to communication requests from a process
pr_init	<code>udp_init</code>	initialization for UDP
pr_fasttimo	<code>0</code>	not used by UDP
pr_slowtimo	<code>0</code>	not used by UDP
pr_drain	<code>0</code>	not used by UDP
pr_sysctl	<code>udp_sysctl</code>	for <code>sysctl(8)</code> system call

We describe the five functions that begin

with `udp_` in this chapter. We also cover a sixth function, `udp_output`, which is not in the protocol switch entry but is called by `udp_usrreq` when a UDP datagram is output.

---

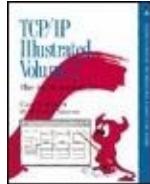
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.4 UDP Header

The UDP header is defined as a `udphdr` structure. [Figure 23.9](#) shows the C structure and [Figure 23.10](#) shows a picture of the UDP header.

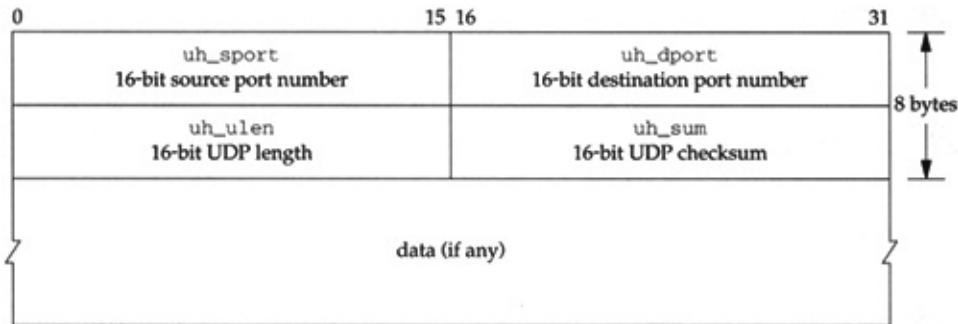
**Figure 23.9. `udphdr` structure.**

```
39 struct udphdr {
40     u_short uh_sport;           /* source port */
41     u_short uh_dport;          /* destination port */
42     short   uh_ulen;           /* udp length */
43     u_short uh_sum;            /* udp checksum */
44 };
```

*udp.h*

*udp.h*

**Figure 23.10. UDP header and optional data.**



In the source code the UDP header is normally referenced as an IP header immediately followed by a UDP header. This is how `udp_input` processes received IP datagrams, and how `udp_output` builds outgoing IP datagrams. This combined IP/UDP header is a `udpipphdr` structure, shown in [Figure 23.11](#).

**Figure 23.11. `udpipphdr` structure: combined IP/UDP header.**

```

38 struct udpiphdr {
39     struct ipovly ui_i;          /* overlaid ip structure */
40     struct udphdr ui_u;         /* udp header */
41 };
42 #define ui_next    ui_i.ih_next
43 #define ui_prev    ui_i.ih_prev
44 #define ui_x1      ui_i.ih_x1
45 #define ui_pr      ui_i.ih_pr
46 #define ui_len     ui_i.ih_len
47 #define ui_src     ui_i.ih_src
48 #define ui_dst     ui_i.ih_dst
49 #define ui_sport   ui_u.uh_sport
50 #define ui_dport   ui_u.uh_dport
51 #define ui_ulen   ui_u.uh_ulen
52 #define ui_sum    ui_u.uh_sum

```

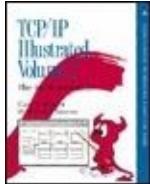
The 20-byte IP header is defined as an ipovly structure, shown in [Figure 23.12](#).

**Figure 23.12. ipovly structure.**

```
38 struct ipovly {                                ip_var.h
39     caddr_t ih_next, ih_prev; /* for protocol sequence q's */
40     u_char ih_x1;           /* (unused) */
41     u_char ih_pr;           /* protocol */
42     short  ih_len;          /* protocol length */
43     struct in_addr ih_src;  /* source internet address */
44     struct in_qaddr ih_dst; /* destination internet address */
45 };
```

ip\_var.h

Unfortunately this structure is not a real IP header, as shown in [Figure 8.8](#). The size is the same (20 bytes) but the fields are different. We'll return to this discrepancy when we discuss the calculation of the UDP checksum in [Section 23.6](#).



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.5 udp\_init Function

The domaininit function calls UDP's initialization function (`udp_init`, [Figure 23.13](#)) at system initialization time.

#### Figure 23.13. `udp_init` function.

```
50 void
51 udp_init()
52 {
53     udb.inp_next = udb.inp_prev = &udb;
54 }
```

The only action performed by this function is to set the next and previous pointers in the head PCB (udb) to point to itself. This is an empty doubly linked list.

The remainder of the udb PCB is initialized to 0, although the only other field used in this head PCB is `inp_lport`, the next UDP ephemeral port number to allocate. In the solution for [Exercise 22.4](#) we mention that because this local port number is initialized to 0, the first ephemeral port number will be 1024.

---

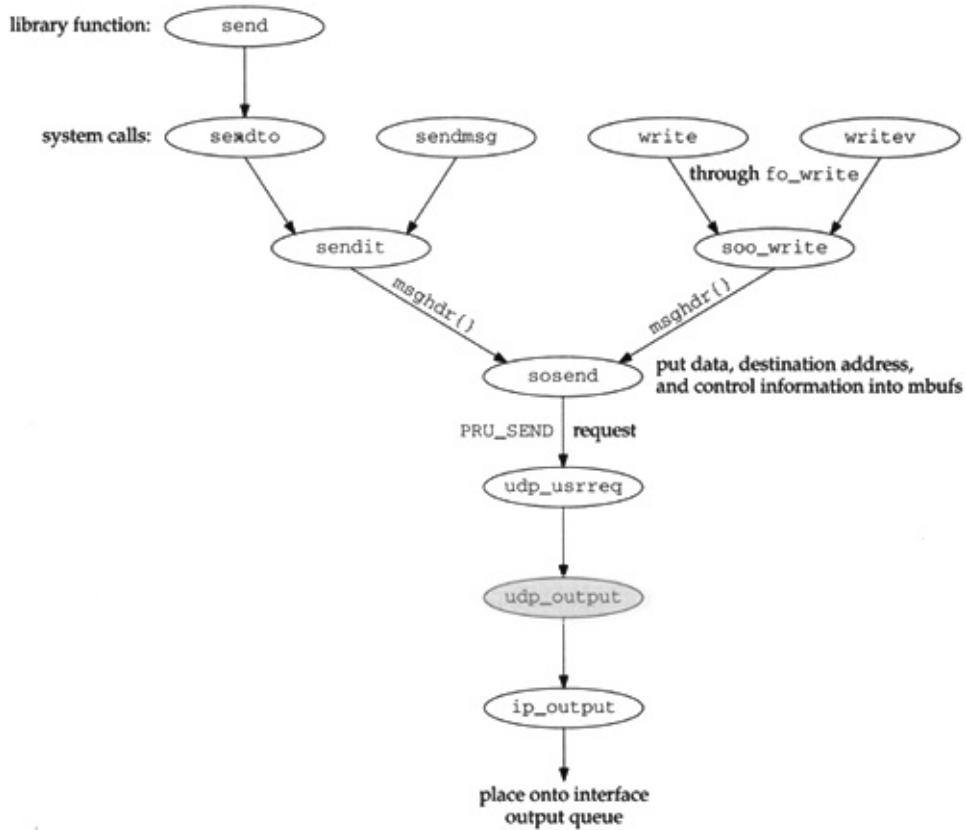
## Chapter 23. UDP: User Datagram Protocol

---

### 23.6 udp\_output Function

UDP output occurs when the application calls one of sendmsg, write, or writev. If the socket is connected, then a destination address must be specified; although a destination address cannot be specified if the socket is unconnected, only sendto and sendmsg can be specified. [Figure 23.14](#) summarizes how these functions are called, which in turn calls ip\_output.

**Figure 23.14. How the five write**



All five functions end up calling `soshnd`, passing  
The data to output is packaged into an mbuf ch  
optional control information are also put into m

UDP calls the function `udp_output`, which we sh  
arguments are `inp`, a pointer to the socket Inte  
output; `addr`, an optional pointer to an mbuf wi  
`sockaddr_in` structure; and `control`, an optional  
`sendmsg`.

**Figure 23.15. `udp_output` function: ter**

---

```

333 int
334 udp_output(inp, m, addr, control)
335 struct inpcb *inp;
336 struct mbuf *m;
337 struct mbuf *addr, *control;
338 {
339     struct udiphdr *ui;
340     int    len = m->m_pkthdr.len;
341     struct in_addr laddr;
342     int    s, error = 0;
343     if (control)
344         m_freem(control);      /* XXX */
345     if (addr) {
346         laddr = inp->inp_laddr;
347         if (inp->inp_faddr.s_addr != INADDR_ANY) {
348             error = EISCONN;
349             goto release;
350         }
351         /*
352          * Must block input while temporarily connected.
353          */
354         s = splnet();
355         error = in_pcbservice(inp, addr);
356         if (error) {
357             splx(s);
358             goto release;
359         }
360     } else {
361         if (inp->inp_faddr.s_addr == INADDR_ANY) {
362             error = ENOTCONN;
363             goto release;
364         }
365     }
366     /*
367      * Calculate data length and get an mbuf for UDP and IP headers.
368      */
369     M_PREPEND(m, sizeof(struct udiphdr), M_DONTWAIT);
370     if (m == 0) {
371         error = ENOBUFS;
372         goto release;
373     }

```

```
/* remainder of function shown in Figure 23.20 */
```

```

409     release:
410     m_freem(m);
411     return (error);
412 }
```

---

*udp\_usrreq.c*

## Discard optional control information

333-344

Any optional control information is discarded by  
does not use control information for any purpose

The comment XXX is because the control information is not yet available. Other protocols, such as the routing domain association, provide their own control information.

## Temporarily connect an unconnected socket

345-359

If the caller specifies a destination address for the socket, it is temporarily connected to that destination address and then disconnected at the end of this function. Before the socket is already connected, and, if so, the function ignores that specifies a destination address on a connected socket.

Before the socket is temporarily connected, IP is not notified of the connection because the temporary connect changes the following address in the socket's PCB. If a received UDP datagram is on a socket that is temporarily connected, that datagram could be processed by the processor priority to splnet only stops a software interrupt. If a software interrupt is executed ([Figure 1.12](#)), it does not prevent the datagrams from being processed and placing them onto IP's input queue.

[[Partridge and Pink 1993](#)] note that this operation is very expensive and consumes nearly one-third of the CPU time.

The local address from the PCB is saved in laddr. If the local wildcard address it will be changed by in\_pcbcc.

The same rules apply to the destination address since `in_pcbconnect` is called for both cases.

360-364

If the process doesn't specify a destination address is returned.

## Prepend IP and UDP headers

366-373

`M_PREPEND` allocates room for the IP and UDP scenario, assuming there is not room in the first byte. [Exercise 23.1](#) details the other possible scenarios: the socket is temporarily connected, IP process

Earlier Berkeley releases incorrectly specified

## Appending IP/UDP Headers and Mbuf Clusters

There is a subtle interaction between the `M_PR` placed into a cluster by `sosend`, then 56 bytes at the beginning of the cluster, allowing room for the `M_PREPEND` from allocating another mbuf just enough `M_LEADINGSPACE` to calculate how much space is available.

```
#define M.LEADINGSPACE(m) \
    ((m)->m_flags & M_EXT ? /* (n
        (m)->m_flags & M_PKTHDR ?
        (m)->m_data - (m)->m_dat)
```

The code that correctly calculates the amount of space needed and the macro always returns 0 if the data is in a cluster, `M_PREPEND` always allocates a new buffer to hold room allocated for this purpose by `sosend`.

The reason for commenting out the correct code is that clusters can be shared ([Section 2.9](#)), and, if it is shared, updating one cluster could wipe out someone else's data.

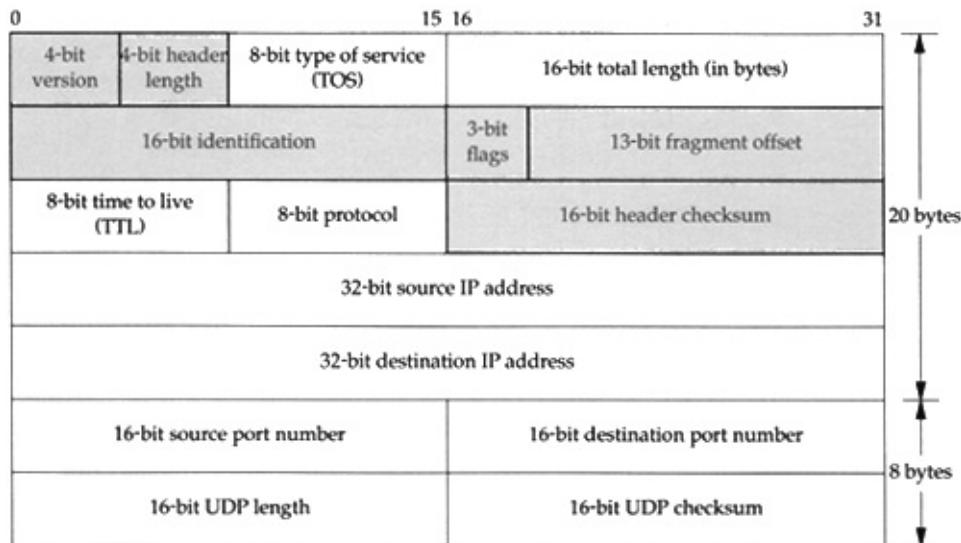
With UDP data, clusters are not shared, since TCP, however, saves a copy of the data in its socket buffer (until acknowledged), and if the data is in a cluster, `M.LEADINGSPACE`, because `sosend` leaves room for the cluster for datagram protocols. `tcp_output` always reserves space for the protocol headers.

## UDP Checksum Calculation and Pseudo-Header

Before showing the last half of `udp_output` we will show how it adds IP/UDP headers, calculates the UDP checksum, and prepares the data for output. The way this is done with the ipovly

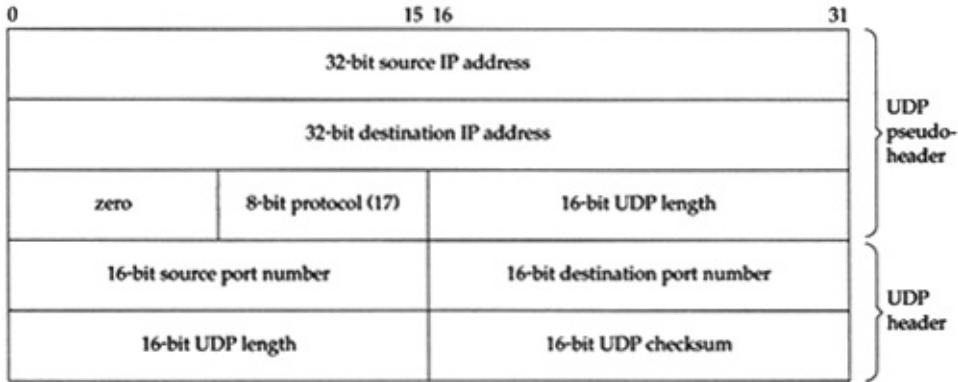
**Figure 23.16** shows the 28-byte IP/UDP header the chain pointed to by m. The unshaded fields are filled in by ip\_output. This figure shows the

### Figure 23.16. IP/UDP headers: unshaded fie



The UDP checksum is calculated over three areas: (1) 12 bytes from the IP header, (2) the 8-byte UDP header, and (3) 8 bytes of pseudo-header used for the checksum calculation. The pseudo-header used for the checksum calculation is identical to the one shown in Figure 23.16.

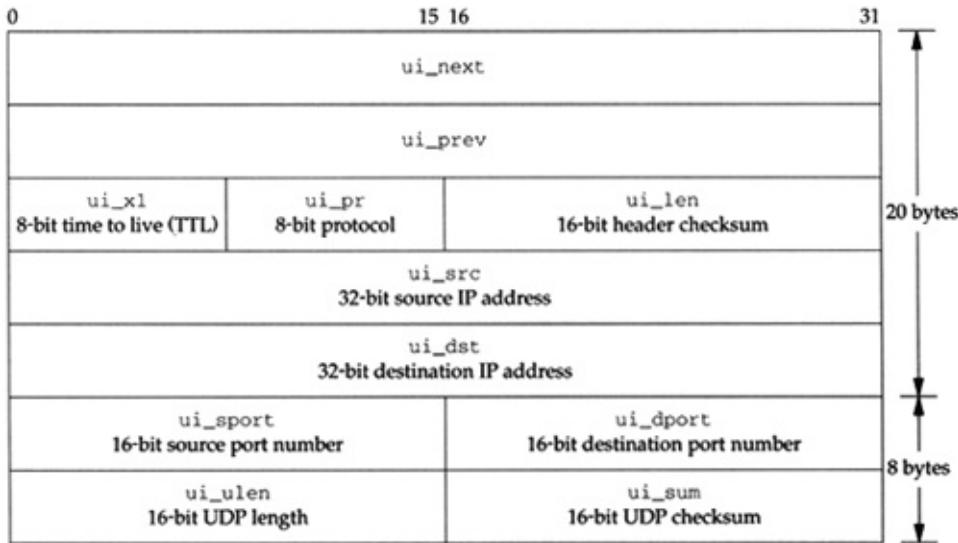
### Figure 23.17. Pseudo-header used for UDP checksum calculation



The following three facts are used in computing the pseudo-header ([Figure 23.17](#)) looks similar [23.16](#)): two 8-bit values and a 16-bit value. (2) pseudo-header is irrelevant. Actually, the comp on the order of the 16-bit values that are used of 0 in the checksum computation has no effect

`udp_output` takes advantage of these three fact ([Figure 23.11](#)), which we depict in [Figure 23.18](#) chain pointed to by the argument `m`.

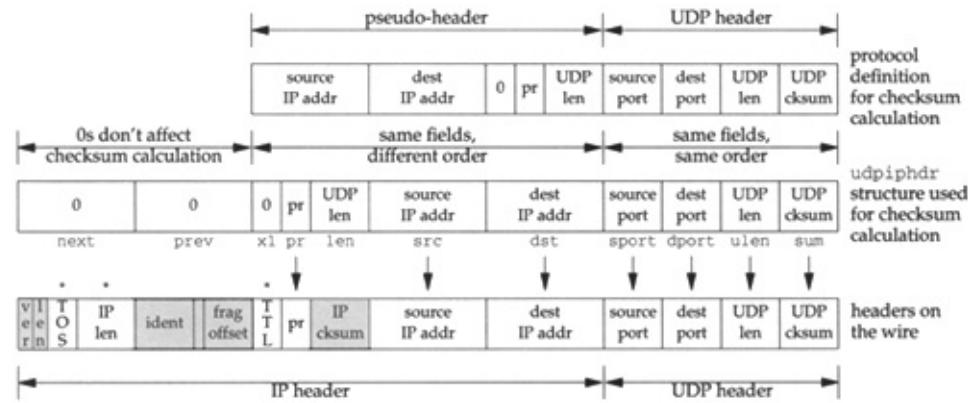
**Figure 23.18. `udpiphdr struct`**



The last three 32-bit words in the 20-byte IP header (ui\_src, ui\_dst, and ui\_len) are used as the pseudo-header for checksum calculation. The first two 32-bit words in the IP header (ui\_next and ui\_prev) are not part of the pseudo-header but they're initialized to 0, and don't affect the checksum calculation.

Figure 23.19 summarizes the operations we've

## Figure 23.19. Operations to fill in IP/U



1. The top picture shown in Figure 23.19 is

**which corresponds to [Figure 23.17](#).**

- The middle picture is the `udphdr` structure to [Figure 23.11](#). (To make the figure readable, This is the structure built by `udp_output` in the checksum.
- The bottom picture shows the IP/UDP headers [Figure 23.16](#). The seven fields with an arrow at checksum computation. The three fields with an arrow at the checksum computation. The remaining six :

[Figure 23.20](#) shows the last half of the `udp_out`

**[Figure 23.20. `udp\_output` function: fill](#)**

---

```

374  /*
375   * Fill in mbuf with extended UDP header
376   * and addresses and length put into network format.
377   */
378  ui = mtod(m, struct udiphdr *);
379  ui->ui_next = ui->ui_prev = 0;
380  ui->ui_x1 = 0;
381  ui->ui_pr = IPPROTO_UDP;
382  ui->ui_len = htons((u_short) len + sizeof(struct udphdr));
383  ui->ui_src = inp->inp_laddr;
384  ui->ui_dst = inp->inp_faddr;
385  ui->ui_sport = inp->inp_lport;
386  ui->ui_dport = inp->inp_fport;
387  ui->ui_ulen = ui->ui_len;

388  /*
389   * Stuff checksum and output datagram.
390   */
391  ui->ui_sum = 0;
392  if (udpcsum) {
393      if ((ui->ui_sum = in_cksum(m, sizeof(struct udiphdr) + len)) == 0)
394          ui->ui_sum = 0xffff;
395  }
396  ((struct ip *) ui)->ip_len = sizeof(struct udiphdr) + len;
397  ((struct ip *) ui)->ip_ttl = inp->inp_ip.ip_ttl; /* XXX */
398  ((struct ip *) ui)->ip_tos = inp->inp_ip.ip_tos; /* XXX */
399  udpstat.udps_opackets++;
400  error = ip_output(m, inp->inp_options, &inp->inp_route,
401                  inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST),
402                  inp->inp_moptions);
403  if (addr) {
404      in_pcbservice(addr);
405      inp->inp_laddr = laddr;
406      splx(s);
407  }
408  return (error);

```

---

udp\_usrreq.c

## Prepare pseudo-header for checksum computation

374-387

All the members in the udiphdr structure (Figure 4-11) except the source and destination IP addresses and port numbers are local and foreign sockets from the PCB are already converted to network byte order. The UDP length field (ui\_len) will be 0 plus the size of the UDP header (8). The checksum calculation: ui\_len and ui\_ulen. One

## Calculate checksum

388-395

The checksum is calculated by first setting it to zero. If checksums are disabled (a bad idea see Section 11.3 of Vol 1), the checksum is 0, 16 one bits are stored in the header. All one bits and all zero bits are both considered valid. A UDP packet without a checksum (the checksum field whose value is 0 (the checksum is 16 one bits))

The variable `udpcksum` ([Figure 23.3](#)) normally contains zero. The kernel can be compiled for 4.2BSD compatibility.

## Fill in UDP length, TTL, and TOS

396-398

The pointer `ui` is cast to a pointer to a standard C structure. The fields are set by UDP. The IP length field is set to the size of the IP/UDP headers. Notice that this field is stored in network byte order like the rest of the multibyte fields. It is converted to network byte order before transmission.

The TTL and TOS fields in the IP header are the same as the values are defaulted by UDP when the socket is created with `setsockopt`. Since these three fields are not used in the UDP checksum computation, they are filled in before `ip_output` is called.

## Send datagram

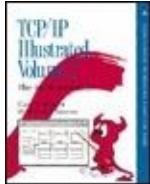
400-402

ip\_output sends the datagram. The second argument is a pointer to a struct ip\_options argument set using setsockopt. These IP options are placed in the packet header. The third argument is a pointer to the cached route in the PCB. The fourth argument is a pointer to the socket options structure. The only socket options that are passed are SO\_DONTROUTE (allow bypassing routing tables) and SO\_BROADCAST (allow broadcast). The fifth argument is a pointer to the multicast options for this socket.

## Disconnect temporarily connected socket

403-407

If the socket was temporarily connected, in\_pcb->lcb->lcb\_ip is restored in the PCB, and the interrupt is disabled.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.7 udp\_input Function

UDP output is driven by a process calling one of the five write functions. The functions shown in [Figure 23.14](#) are all called directly as part of the system call. UDP input, on the other hand, occurs when IP input receives an IP datagram on its input queue whose protocol field specifies UDP. IP calls the function `udp_input` through the `pr_input` function in the protocol switch table ([Figure 8.15](#)). Since IP input is at the software interrupt level, `udp_input` also executes at this level. The goal of `udp_input` is to place the UDP datagram onto the appropriate socket's buffer and wake up any process blocked

for input on that socket.

We'll divide our discussion of the `udp_input` function into three sections:

## **1. the general validation that UDP performs on the received datagram,**

- processing UDP datagrams destined for a unicast address: locating the appropriate PCB and placing the datagram onto the socket's buffer, and
- processing UDP datagrams destined for a broadcast or multicast address: the datagram may be delivered to multiple sockets.

This last step is new with the support of multicasting in Net/3, but consumes almost one-third of the code.

## **General Validation of Received UDP Datagram**

[Figure 23.21](#) shows the first section of UDP input.

## Figure 23.21. udp\_input function: general validation of received UDP datagram.

```
55 void
56 udp_input(m, iphlen)
57 struct mbuf *m;
58 int     iphlen;
59 {
60     struct ip *ip;
61     struct udphdr *uh;
62     struct inpcb *inp;
63     struct mbuf *opts = 0;
64     int     len;
65     struct ip save_ip;
66     udpstat.udps_ipackets++;

67     /*
68      * Strip IP options, if any; should skip this,
69      * make available to user, and use on returned packets,
70      * but we don't yet have a way to check the checksum
71      * with options still present.
72     */
73     if (iphlen > sizeof(struct ip)) {
74         ip_strioptions(m, (struct mbuf *) 0);
75         iphlen = sizeof(struct ip);
76     }
77     /*
78      * Get IP and UDP header together in first mbuf.
79     */
80     ip = mtod(m, struct ip *);
81     if (m->m_len < iphlen + sizeof(struct udphdr)) {
82         if ((m = m_pullup(m, iphlen + sizeof(struct udphdr))) == 0) {
83             udpstat.udps_hdrops++;
84             return;
85         }
86         ip = mtod(m, struct ip *);
87     }
88     uh = (struct udphdr *) ((caddr_t) ip + iphlen);

89     /*
90      * Make mbuf data length reflect UDP length.
91      * If not enough data to reflect UDP length, drop.
92     */
93     len = ntohs((u_short) uh->uh_ulen);
94     if (ip->ip_len != len) {
95         if (len > ip->ip_len) {
96             udpstat.udps_badlen++;
97             goto bad;
98         }
99         m_adj(m, len - ip->ip_len);
100        /* ip->ip_len = len; */
101    }
102    /*
103     * Save a copy of the IP header in case we want to restore
104     * it for sending an ICMP error message in response.
105     */
106    save_ip = *ip;
```

```
107  /*
108   * Checksum extended UDP header and data.
109   */
110  if (udpcksum && uh->uh_sum) {
111      ((struct ipovly *) ip)->ih_next = 0;
112      ((struct ipovly *) ip)->ih_prev = 0;
113      ((struct ipovly *) ip)->ih_x1 = 0;
114      ((struct ipovly *) ip)->ih_len = uh->uh_ulen;
115      if (uh->uh_sum == in_cksum(m, len + sizeof(struct ip))) {
116          udpstat.udps_badsum++;
117          m_freem(m);
118          return;
119      }
120  }
```

---

*udp\_usrreq.c*

## 55-65

The two arguments to `udp_input` are `m`, a pointer to an mbuf chain containing the IP datagram, and `iphlen`, the length of the IP header (including possible IP options).

## Discard IP options

## 67-76

If IP options are present they are discarded by `ip_strioptions`. As the comments indicate, UDP should save a copy of the IP options and make them available to the receiving process through the `IP_RECVOPTS` socket option, but this isn't implemented yet.

## 77-88

If the length of the first mbuf on the mbuf chain is less than 28 bytes (the size of the IP header plus the UDP header), m\_pullup rearranges the mbuf chain so that at least 28 bytes are stored contiguously in the first mbuf.

## Verify UDP length

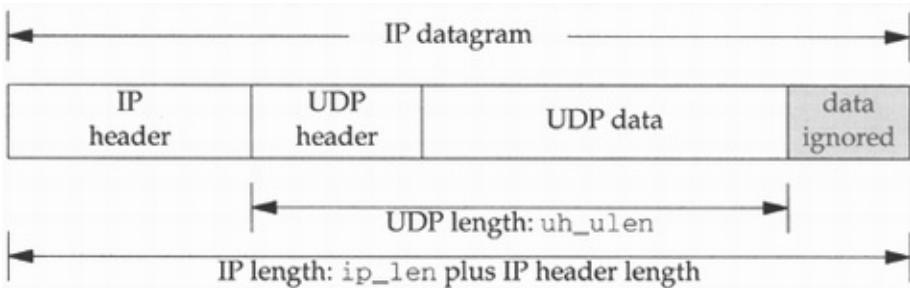
89-101

There are two lengths associated with a UDP datagram: the length field in the IP header (ip\_len) and the length field in the UDP header (uh\_ulen). Recall that ipintr subtracted the length of the IP header from ip\_len before calling udp\_input ([Figure 10.11](#)). The two lengths are compared and there are three possibilities:

**1. ip\_len equals uh\_ulen. This is the common case.**

- ip\_len is greater than uh\_ulen. The IP datagram is too big, as shown in [Figure 23.22](#).

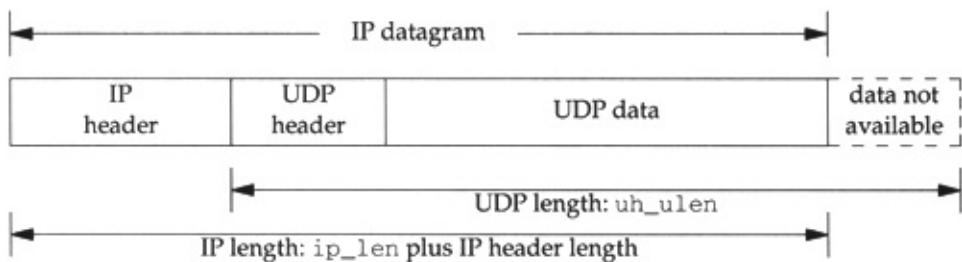
## Figure 23.22. UDP length too small.



The code believes the smaller of the two lengths (the UDP header length) and `m_adj` removes the excess bytes of data from the end of the datagram. In the code the second argument to `m_adj` is negative, which we said in [Figure 2.20](#) trims data from the end of the mbuf chain. It is possible in this scenario that the UDP length field has been corrupted. If so, the datagram will probably be discarded shortly, assuming the sender calculated the UDP checksum, that this checksum detects the error, and that the receiver verifies the checksum. The IP length field should be correct since it was verified by IP against the amount of data received from the interface, and the IP length field is covered by the mandatory IP header checksum.

- `ip_len` is less than `uh_ulen`. The IP datagram is smaller than possible, given the length in the UDP header. [Figure 23.23](#) shows this case.

**Figure 23.23. UDP length too big.**



Something is wrong and the datagram is discarded. There is no other choice here: if the UDP length field has been corrupted, it can't be detected with the UDP checksum. The correct UDP length is needed to calculate the checksum.

As we've said, the UDP length is redundant. In [Chapter 28](#) we'll see that TCP does not have a length field in its header; it uses the IP length field, minus the lengths of the IP and TCP headers, to determine the amount of data in the datagram. Why does the UDP length

field exist? Possibly to add a small amount of error checking, since UDP checksums are optional.

## Save copy of IP header and verify UDP checksum

102-106

udp\_input saves a copy of the IP header before verifying the checksum, because the checksum computation wipes out some of the fields in the original IP header.

110

The checksum is verified only if UDP checksums are enabled for the kernel (udpcksum), and if the sender calculated a UDP checksum (the received checksum is nonzero).

This test is incorrect. If the sender calculated a checksum, it should be verified, regardless of whether outgoing checksums are calculated or not. The variable udpcksum should only specify whether outgoing checksums are

calculated. Unfortunately many vendors have copied this incorrect test, although many vendors today finally ship their kernels with UDP checksums enabled by default.

### 111-120

Before calculating the checksum, the IP header is referenced as an ipovly structure ([Figure 23.18](#)) and the fields are initialized as described in the previous section when the UDP checksum is calculated by `udp_output`.

At this point special code is executed if the datagram is destined for a broadcast or multicast IP address. We defer this code until later in the section.

## Demultiplexing Unicast Datagrams

Assuming the datagram is destined for a unicast address, [Figure 23.24](#) shows the code that is executed.

### Figure 23.24. `udp_input` function:

# demultiplex unicast datagram.

---

udp\_usrreq.c

```
/* demultiplex broadcast & multicast datagrams (Figure 23.26) */

206  /*
207   * Locate pcb for unicast datagram.
208   */
209  inp = udp_last_inpcb;
210  if (inp->inp_lport != uh->uh_dport ||
211      inp->inp_fport != uh->uh_sport ||
212      inp->inp_faddr.s_addr != ip->ip_src.s_addr ||
213      inp->inp_laddr.s_addr != ip->ip_dst.s_addr) {
214      inp = in_pcblookup(&udb, ip->ip_src, uh->uh_sport,
215                          ip->ip_dst, uh->uh_dport, INPLOOKUP_WILDCARD);
216      if (inp)
217          udp_last_inpcb = inp;
218      udpstat.udpps_pcbaclmiss++;
219  }
220  if (inp == 0) {
221      udpstat.udpps_noport++;
222      if (m->m_flags & (M_BCAST | M_MCAST)) {
223          udpstat.udpps_noportbcast++;
224          goto bad;
225      }
226      *ip = save_ip;
227      ip->ip_len += iphlen;
228      icmp_error(m, ICMP_UNREACH, ICMP_UNREACH_PORT, 0, 0);
229      return;
230 }
```

---

udp\_usrreq.c

## Check one-behind cache

206-209

UDP maintains a pointer to the last Internet PCB for which it received a datagram, `udp_last_inpcb`. Before calling `in_pcblookup`, which might have to search many PCBs on the UDP list, the foreign and local addresses and ports of that last PCB are compared against the received

datagram. This is called a *one-behind cache* [Partridge and Pink 1993], and it is based on the assumption that the next datagram received has a high probability of being destined for the same socket as the last received datagram [Mogul 1991]. This cache was introduced with the 4.3BSD Tahoe release.

## 210-213

The order of the four comparisons between the cached PCB and the received datagram is intentional. If the PCBs don't match, the comparisons should stop as soon as possible. The highest probability is that the destination port numbers are different; this is therefore the first test. The lowest probability of a mismatch is between the local addresses, especially on a host with just one interface, so this is the last test.

Unfortunately this one-behind cache, as coded, is practically useless [Partridge and Pink 1993]. The most common type of UDP server binds only its well-known port, leaving its local address, foreign address, and foreign port wildcarded. The most

common type of UDP client does not connect its UDP socket; it specifies the destination address for each datagram using sendto. Therefore most of the time the three values in the PCB inp\_laddr, inp\_faddr, and inp\_fport are wildcards. In the cache comparison the four values in the received datagram are never wildcards, meaning the cache entry will compare equal with the received datagram only when the PCB has all four local and foreign values specified to nonwildcard values. This happens only for a connected UDP socket.

On the system bsdi, the counter udpps\_pcbcachemiss was 41,253 and the counter udps\_ipackets was 42,485. This is less than a 3% cache hit rate.

The netstat -s command prints most of the fields in the udpstat structure ([Figure 23.5](#)). Unfortunately the Net/3 version, and most vendor's versions, never print udpps\_pcbcachemiss. If you want to see the value, use a debugger to examine the variable in the running kernel.

## Search all UDP PCBs

214-218

Assuming the comparison with the cached PCB fails, `in_pcblkup` searches for a match. The `INPLOOKUP_WILDCARD` flag is specified, allowing a wildcard match. If a match is found, the pointer to the PCB is saved in `udp_last_inpcb`, which we said is a cache of the last received UDP datagram's PCB.

## Generate ICMP port unreachable error

220-230

If a matching PCB is not found, UDP normally generates an ICMP port unreachable error. First the `m_flags` for the received mbuf chain is checked to see if the datagram was sent to a link-level broadcast or multicast destination address. It is possible to receive an IP datagram with a unicast IP address that was sent to a broadcast or multicast link-level address, but an ICMP port unreachable error must

not be generated. If it is OK to generate the ICMP error, the IP header is restored to its received value (`save_ip`) and the IP length is also set back to its original value.

This check for a link-level broadcast or multicast address is redundant. `icmp_error` also performs this check. The only advantage in this redundant check is to maintain the counter `udps_noportbcst` in addition to the counter `udps_noport`.

The addition of `iphlen` back into `ip_len` is a bug. `icmp_error` will also do this, causing the IP length field in the IP header returned in the ICMP error to be 20 bytes too large. You can tell if a system has this bug by adding a few lines of code to the Traceroute program (Chapter 8 of Volume 1) to print this field in the ICMP port unreachable that is returned when the destination host is finally reached.

[Figure 23.25](#) is the next section of processing for a unicast datagram, delivering the datagram to the socket

corresponding to the destination PCB.

## Figure 23.25. udp\_input function: deliver unicast datagram to socket.

```
----- udp_usrreq.c
231  /*
232   * Construct sockaddr format source address.
233   * Stuff source address and datagram in user buffer.
234   */
235  udp_in.sin_port = uh->uh_sport;
236  udp_in.sin_addr = ip->ip_src;

237  if (inp->inp_flags & INP_CONTROLOPTS) {
238      struct mbuf **mp = &opts;

239      if (inp->inp_flags & INP_RECVDSTADDR) {
240          *mp = udp_saveopt((caddr_t) & ip->ip_dst,
241                             sizeof(struct in_addr), IP_RECVDSTADDR);
242          if (*mp)
243              mp = &(*mp)->m_next;
244      }
245 #ifdef notyet
246      /* IP options were tossed above */
247      if (inp->inp_flags & INP_RECVOPTS) {
248          *mp = udp_saveopt((caddr_t) opts_deleted_above,
249                            sizeof(struct in_addr), IP_RECVOPTS);
250          if (*mp)
251              mp = &(*mp)->m_next;
252      }
253      /* ip_srcroute doesn't do what we want here, need to fix */
254      if (inp->inp_flags & INP_RECVRETOPTS) {
255          *mp = udp_saveopt((caddr_t) ip_srcroute(),
256                            sizeof(struct in_addr), IP_RECVRETOPTS);
257          if (*mp)
258              mp = &(*mp)->m_next;
259      }
260 #endif
261  }
262  iphlen += sizeof(struct udphdr);
263  m->m_len -= iphlen;
264  m->m_pkthdr.len -= iphlen;
265  m->m_data += iphlen;
266  if (sbappendaddr(&inp->inp_socket->so_rcv, (struct sockaddr *) &udp_in,
267                   m, opts) == 0) {
268      udpstat.udps_fullsock++;
269      goto bad;
270  }
271  sorwakeup(inp->inp_socket);
272  return;

273 bad:
274     m_freem(m);
275     if (opts)
276         m_freem(opts);
277 }
```

----- udp\_usrreq.c

## **Return source IP address and source port**

231-236

The source IP address and source port number from the received IP datagram are stored in the global sockaddr\_in structure udp\_in. This structure is passed as an argument to sbappendaddr later in the function.

Using a global to hold the IP address and port number is OK because udp\_input is single threaded. When this function is called by ipintr it processes the received datagram completely before returning. Also, sbappendaddr copies the socket address structure from the global into an mbuf.

## **IP\_RECVSTADDR socket option**

237-244

The constant INP\_CONTROLOPTS is the combination of the three socket options

that the process can set to cause control information to be returned through the `recvmsg` system call for a UDP socket ([Figure 22.5](#)). The `IP_RECVDSTADDR` socket option returns the destination IP address from the received UDP datagram as control information. The function `udp_saveopt` allocates an mbuf of type `MT_CONTROL` and stores the 4-byte destination IP address in the mbuf. We show this function in [Section 23.8](#).

This socket option appeared with 4.3BSD Reno and was intended for applications such as TFTP, the Trivial File Transfer Protocol, that should not respond to client requests that are sent to a broadcast address. Unfortunately, even if the receiving application uses this option, it is nontrivial to determine if the destination IP address is a broadcast address or not ([Exercise 23.6](#)).

When the multicasting changes were added in 4.4BSD, this code was left in only for datagrams destined for a unicast address. We'll see in [Figure](#)

[23.26](#) that this option is not implemented for datagrams sent to a broadcast or multicast address. This defeats the purpose of the option!

**Figure 23.26. `udp_input` function:  
demultiplexing of broadcast and  
multicast datagrams.**

```

121      if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr)) ||
122          in_broadcast(ip->ip_dst, m->m_pkthdr.rcvif)) {
123          struct socket *last;
124          /*
125           * Deliver a multicast or broadcast datagram to *all* sockets
126           * for which the local and remote addresses and ports match
127           * those of the incoming datagram. This allows more than
128           * one process to receive multi/broadcasts on the same port.
129           * (This really ought to be done for unicast datagrams as
130           * well, but that would cause problems with existing
131           * applications that open both address-specific sockets and
132           * a wildcard socket listening to the same port -- they would
133           * end up receiving duplicates of every unicast datagram.
134           * Those applications open the multiple sockets to overcome an
135           * inadequacy of the UDP socket interface, but for backwards
136           * compatibility we avoid the problem here rather than
137           * fixing the interface. Maybe 4.5BSD will remedy this?)
138           */
139
140          /*
141           * Construct sockaddr format source address.
142           */
143          udp_in.sin_port = uh->uh_sport;
144          udp_in.sin_addr = ip->ip_src;
145          m->m_len -= sizeof(struct udpiphdr);
146          m->m_data += sizeof(struct udpiphdr);
147          /*
148           * Locate pcb(s) for datagram.
149           * (Algorithm copied from raw_intr().)
150           */
151          last = NULL;
152          for (inp = udb.inp_next; inp != &udb; inp = inp->inp_next) {
153              if (inp->inp_lport != uh->uh_dport)
154                  continue;
155              if (inp->inp_laddr.s_addr != INADDR_ANY) {
156                  if (inp->inp_laddr.s_addr !=
157                      ip->ip_dst.s_addr)
158                      continue;
159                  if (inp->inp_faddr.s_addr != INADDR_ANY) {
160                      if (inp->inp_faddr.s_addr !=
161                          ip->ip_src.s_addr ||
162                          inp->inp_fport != uh->uh_sport)
163                          continue;
164                  }
165                  if (last != NULL) {
166                      struct mbuf *n;
167
168                      if ((n = m_copy(m, 0, M_COPYALL)) != NULL) {
169                          if (sbappendaddr(&last->so_rcv,
170                                           (struct sockaddr *) &udp_in,
171                                           n, (struct mbuf *) 0) == 0) {
172                              m_free(n);
173                              udpstat.udps_fullsock++;
174                          } else
175                              sorwakeup(last);
176                      }
177                  last = inp->inp_socket;
178                  /*
179                   * Don't look for additional matches if this one does
180                   * not have either the SO_REUSEPORT or SO_REUSEADDR
181                   * socket options set. This heuristic avoids searching
182                   * through all pcbs in the common case of a non-shared
183                   * port. It assumes that an application will never
184                   * clear these options after setting them.
185                   */
186                  if ((last->so_options & (SO_REUSEPORT | SO_REUSEADDR) == 0))
187                      break;
188              }
189              if (last == NULL) {
190                  /*
191                   * No matching pcb found; discard datagram.
192                   * (No need to send an ICMP Port Unreachable
193                   * for a broadcast or multicast datgram.)
194                   */
195                  udpstat.udps_noportbcast++;
196                  goto bad;
197              }
198              if (sbappendaddr(&last->so_rcv, (struct sockaddr *) &udp_in,
199                           m, (struct mbuf *) 0) == 0) {
200                  udpstat.udps_fullsock++;
201                  goto bad;
202              }
203              sorwakeup(last);
204          }
205      }

```

----- udp\_usrreq.c -----

## Unimplemented socket options

245-260

This code is commented out because it doesn't work. The intent of the IP\_RECVOPTS socket option is to return the IP options from the received datagram as control information, and the intent of IP\_RECVRETOPTS socket option is to return source route information. The manipulation of the mp variable by all three IP\_RECV socket options is to build a linked list of up to three mbufs that are then placed onto the socket's buffer by sbappendaddr. The code shown in [Figure 23.25](#) only returns one option as control information, so the m\_next pointer of that mbuf is always a null pointer.

## Append data to socket's receive queue

262-272

At this point the received datagram (the mbuf chain pointed to by m), is ready to

be placed onto the socket's receive queue along with a socket address structure representing the sender's IP address and port (udp\_in), and optional control information (the destination IP address, the mbuf pointed to by opts). This is done by sbappendaddr. Before calling this function, however, the pointer and lengths of the first mbuf on the chain are adjusted to ignore the IP and UDP headers. Before returning, sorwakeup is called for the receiving socket to wake up any processes asleep on the socket's receive queue.

## Error return

273-276

If an error is encountered during UDP input processing, udp\_input jumps to the label bad. The mbuf chain containing the datagram is released, along with the mbuf chain containing any control information (if present).

## Demultiplexing Multicast and Broadcast Datagrams

We now return to the portion of `udp_input` that handles datagrams sent to a broadcast or multicast IP address. The code is shown in [Figure 23.26](#).

**121-138**

As the comments indicate, these datagrams are delivered to *all* sockets that match, not just a single socket. The inadequacy of the UDP interface that is mentioned refers to the inability of a process to receive asynchronous errors on a UDP socket (notably ICMP port unreachables) unless the socket is connected. We described this in [Section 22.11](#).

**139-145**

The source IP address and port number are saved in the global `sockaddr_in` structure `udp_in`, which is passed to `sbappendaddr`. The mbuf chain's length and data pointer are updated to ignore the IP and UDP headers.

**146-164**

The large for loop scans each UDP PCB to find all matching PCBs. `in_pcblockup` is not called for this demultiplexing because it returns only one PCB, whereas the broadcast or multicast datagram may be delivered to more than one PCB.

If the local port in the PCB doesn't match the destination port from the received datagram, the entry is ignored. If the local address in the PCB is not the wildcard, it is compared to the destination IP address and the entry is skipped if they're not equal. If the foreign address in the PCB is not a wildcard, it is compared to the source IP address and if they match, the foreign port must also match the source port. This last test assumes that if the socket is connected to a foreign IP address it must also be connected to a foreign port, and vice versa. This is the same logic we saw in `in_pcblockup`.

165-177

If this is not the first match found (last is nonnull), a copy of the datagram is placed onto the receive queue for the previous

match. Since sbappendaddr releases the mbuf chain when it is done, a copy is first made by m\_copy. Any processes waiting for this data are awakened by sorwakeups. A pointer to this matching socket structure is saved in last.

This use of the variable last avoids calling m\_copy (an expensive operation since an entire mbuf chain is copied) unless there are multiple recipients for a given datagram. In the common case of a single recipient, the for loop just sets last to the single matching PCB, and when the loop terminates, sbappendaddr places the mbuf chain onto the socket's receive queuea copy is not made.

178-188

If this matching socket doesn't have either the SO\_REUSEPORT or the SO\_REUSEADDR socket option set, then there's no need to check for additional matches and the loop is terminated. The datagram is placed onto the single socket's receive queue in the call to sbappendaddr outside the loop.

**189-197**

If last is null at the end of the loop, no matches were found. An ICMP error is not generated because the datagram was sent to a broadcast or multicast IP address.

**198-204**

The final matching entry (which could be the only matching entry) has the original datagram (m) placed onto its receive queue. After sorwakeup is called, udp\_input returns, since the processing the broadcast or multicast datagram is complete.

The remainder of the function (shown previously in [Figure 23.24](#)) handles unicast datagrams.

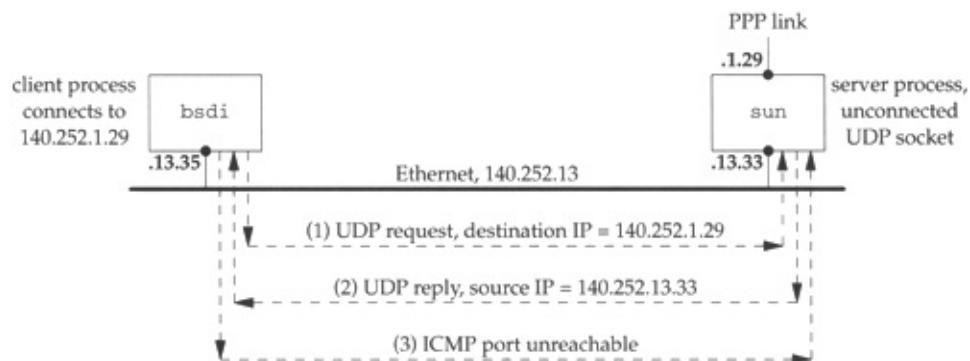
## **Connected UDP Sockets and Multihomed Hosts**

There is a subtle problem when using a connected UDP socket to exchange datagrams with a process on a multihomed host. Datagrams from the peer may arrive

with a different source IP address and will not be delivered to the connected socket.

Consider the example shown in [Figure 23.27](#).

**Figure 23.27. Example of connected UDP socket sending datagram to a multihomed host.**



Three steps take place.

- 1. The client on bsdi creates a UDP socket and connects it to 140.252.1.29, the PPP interface on sun, not the Ethernet interface. A datagram is sent on the socket to the server.**

**The server on sun receives the**

**datagram and accepts it, even though it arrives on an interface that differs from the destination IP address. (sun is acting as a router, so whether it implements the weak end system model or the strong end system model doesn't matter.) The datagram is delivered to the server, which is waiting for client requests on an unconnected UDP socket.**

- The server sends a reply, but since the reply is being sent on an unconnected UDP socket, the source IP address for the reply is chosen by the kernel based on the outgoing interface (140.252.13.33). The destination IP address in the request is not used as the source address for the reply.

When the reply is received by bsdi it is not delivered to the client's connected UDP socket since the IP addresses don't match.

- bsdi generates an ICMP port unreachable error since the reply can't be demultiplexed. (This assumes that there is not another process on bsdi eligible to receive the datagram.)

The problem in this example is that the server does not use the destination IP address from the request as the source IP address of the reply. If it did, the problem wouldn't exist, but this solution is nontrivial see [Exercise 23.10](#). We'll see in [Figure 28.16](#) that a TCP server uses the destination IP address from the client as the source IP address from the server, if the server has not explicitly bound a local IP address to its socket.

---



## Chapter 23. UDP: User Datagram Protocol

---

### 23.8 udp\_saveopt Function

If a process specifies the IP\_RECVDSTADDR socket option, the address from the received datagram `udp_saveopt`

```
*mp = udp_saveopt((caddr_t) &ip->  
                   IP_RECVDSTADDR)
```

Figure 23.28 shows this function.

**Figure 23.28. udp\_saveopt function: creation**

```
278 /*  
279 * Create a "control" mbuf containing the specified data  
280 * with the specified type for presentation with a datagram.  
281 */  
282 struct mbuf *  
283 udp_saveopt(p, size, type)  
284 caddr_t p;  
285 int size;  
286 int type;  
287 {  
288     struct cmsghdr *cp;  
289     struct mbuf *m;  
290     if ((m = m_get(M_DONTWAIT, MT_CONTROL)) == NULL)  
291         return ((struct mbuf *) NULL);  
292     cp = (struct cmsghdr *) mtod(m, struct cmsghdr *);  
293     bcopy(p, CMSG_DATA(cp), size);  
294     size += sizeof(*cp);  
295     m->m_len = size;  
296     cp->cmsg_len = size;  
297     cp->cmsg_level = IPPROTO_IP;  
298     cp->cmsg_type = type;  
299     return (m);  
300 }
```

— *udp\_usrreq.c*

**278-289**

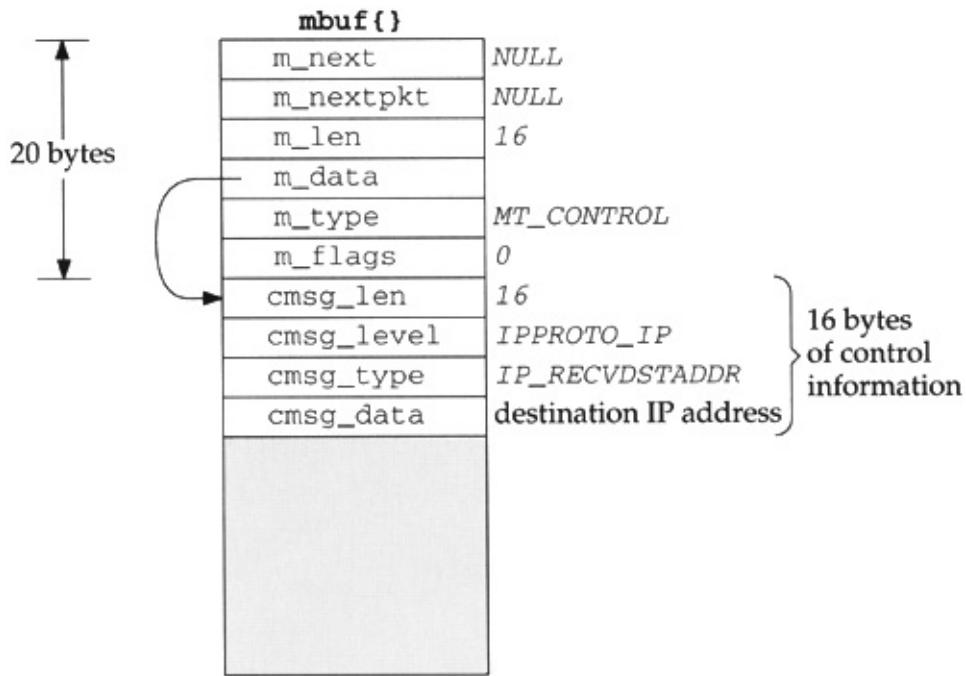
The arguments are *p*, a pointer to the information (for example, the destination IP address from the received datagram); *size*, the size of an IP address); and *type*, (IP\_RECVDSTADDR).

**290-299**

An mbuf is allocated, and since the code is executing in user space, M\_DONTWAIT is specified. The pointer *cp* pointing to the data portion of the mbuf is cast into a pointer to a cmsghdr structure (Figure 23.29). The *m\_len* field of the mbuf is then set (to 16 in this example), and the *cmsg\_len* field of the cmsghdr structure. Figure 23.29 shows the final state of the mbuf.

**Figure 23.29. Mbuf containing destination**

## control information



The `cmsg_len` field contains the length of the control information (16 bytes), and the `cmsg_data` field (4 for this example). If the control information, it must go through the `cmsg_level`, `cmsg_type`, and length of the `cmsg_data` field.

---

## Chapter 23. UDP: User Datagram Protocol

---

### 23.9 udp\_ctlinput Function

When icmp\_input receives an ICMP error (destination unreachable, redirect, source quench, and time exceeded) the function is called:

```
if (ctlfunc = inetsw[ ip_protos[  
    (*ctlfunc) (code, (struct soc
```

For UDP, Figure 22.32 showed that the function is the function in Figure 23.30.

**Figure 23.30. udp\_ctlinput function**

---

```

314 void
315 udp_ctlinput(cmd, sa, ip)
316 int cmd;
317 struct sockaddr *sa;
318 struct ip *ip;
319 {
320     struct udphdr *uh;
321     extern struct in_addr zeroin_addr;
322     extern u_char inetctllerrmap[];
323     if (!PRC_IS_REDIRECT(cmd) &&
324         ((unsigned) cmd >= PRC_NCMDS || inetctllerrmap[cmd] == 0))
325         return;
326     if (ip) {
327         uh = (struct udphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
328         in_pcbsnotify(&udb, sa, uh->uh_dport, ip->ip_src, uh->uh_sport,
329                       cmd, udp_notify);
330     } else
331         in_pcbsnotify(&udb, sa, 0, zeroin_addr, 0, cmd, udp_notify);
332 }

```

---

## 314-322

The arguments are cmd, one of the PRC\_xxx codes, a sockaddr\_in structure containing the source IP address, a pointer to the IP header that caused the error, and a pointer to the UDP header. The cmd parameter problem, source quench, and time exceeded errors all point to the IP header that caused the error. But when udp\_ctlinput is called (Figure 22.32), sa points to a sockaddr\_in structure that should be redirected, and ip is a null pointer. This is the final case, since we saw in Section 22.11 that a socket connected to the destination address. There is however, for other errors, such as a port unreachable error, the IP header contains the unreachable port.

## 323-325

If the error is not a redirect, and either the PRC code or the code in the global array inetctllerrmap, the ICMP message is generated. In this case, we need to review what happens to a received

## **1. icmp\_input converts the ICMP type and**

- The PRC\_xxx error code is passed to the protocol's error handling function.
- The Internet protocols (TCP and UDP) map their errno values using inetctllerrmap, and this value is passed to the error handling function.

Figures 11.1 and 11.2 summarize this process in detail.

Returning to Figure 23.30, we can see what happens when an ICMP error arrives in response to a UDP datagram. icmp\_input sees that the error is PRC\_QUENCH and udp\_ctlinput is called. If the error is blank in Figure 11.2, the error is ignored.

326-331

The function in\_pcbnotify notifies the appropriate PCB. If the final argument to udp\_ctlinput is nonnull, the source address and the datagram that caused the error are passed to in\_pcbnotify. The source address is used to determine the destination address.

## **udp\_notify Function**

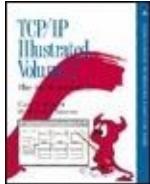
The final argument to in\_pcbnotify is a pointer to the PCB that is to receive the error. The function flowchart is shown in Figure 23.31.

## Figure 23.31. udp\_notify function: notif

```
----- udp_usrreq.c
305 static void
306 udp_notify(inp, errno)
307 struct inpcb *inp;
308 int     errno;
309 {
310     inp->inp_socket->so_error = errno;
311     sorwakeup(inp->inp_socket);
312     sowakeup(inp->inp_socket);
313 }
----- udp_usrreq.c
```

301-313

The `errno` value, the second argument to this function, is stored in the `so_error` variable. By setting this socket variable, the socket will be made ready to receive a process calls `select`. Any processes waiting to receive data on the socket will be awakened to receive the error.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.10 udp\_usrreq Function

The protocol's user-request function is called for a variety of operations. We saw in [Figure 23.14](#) that a call to any one of the five write functions on a UDP socket ends up calling UDP's user-request function with a request of PRU\_SEND.

[Figure 23.32](#) shows the beginning and end of `udp_usrreq`. The body of the switch is discussed in separate figures following this figure. The function arguments are described in [Figure 15.17](#).

### Figure 23.32. Body of `udp_usrreq`

## function.

```
417 int udp_usrreq.c
418 udp_usrreq(so, req, m, addr, control)
419 struct socket *so;
420 int     req;
421 struct mbuf *m, *addr, *control;
422 {
423     struct inpcb *inp = sotoinpcb(so);
424     int     error = 0;
425     int     s;
426
427     if (req == PRU_CONTROL)
428         return (in_control(so, (int) m, (caddr_t) addr,
429                             (struct ifnet *) control));
430     if (inp == NULL && req != PRU_ATTACH) {
431         error = EINVAL;
432         goto release;
433     }
434     /*
435      * Note: need to block udp_input while changing
436      * the udp pcb queue and/or pcb addresses.
437      */
438     switch (req) {
439
440         /* switch cases */
441
442
443
444
445
446
447
448
449
450
451
452     default:
453         panic("udp_usrreq");
454     }
455     release:
456     if (control) {
457         printf("udp control data unexpectedly retained\n");
458         m_freem(control);
459     }
460     if (m)
461         m_freem(m);
462     return (error);
463 }
```

417-428

The PRU\_CONTROL request is from the ioctl system call. The function in\_control processes the request completely.

429-432

The socket pointer was converted to the PCB pointer when inp was declared at the beginning of the function. The only time a null PCB pointer is allowed is when a new socket is being created (PRU\_ATTACH).

433-436

The comment indicates that whenever entries are being added to or deleted from UDP's PCB list, the code must be protected by splnet. This is done because `udp_usrreq` is called as part of a system call, and it doesn't want to be interrupted by UDP input (called by IP input, which is called as a software interrupt) while it is modifying the doubly linked list of PCBs. UDP input is also blocked while modifying the local or foreign addresses or ports in a PCB, to prevent a received UDP datagram from being delivered incorrectly by `in_pcblockup`.

We now discuss the individual case statements. The PRU\_ATTACH request, shown in [Figure 23.33](#), is from the socket system call.

## Figure 23.33. udp\_usrreq function: PRU\_ATTACH and PRU\_DETACH requests.

```
438     case PRU_ATTACH:
439         if (inp != NULL) {
440             error = EINVAL;
441             break;
442         }
443         s = splnet();
444         error = in_pcalloc(so, &udb);
445         splx(s);
446         if (error)
447             break;
448         error = soreserve(so, udp_sendspace, udp_recvspace);
449         if (error)
450             break;
451         ((struct inpcb *) so->so_pcb)->inp_ip.ip_ttl = ip_defttl;
452         break;
453     case PRU_DETACH:
454         udp_detach(inp);
455         break;
```

438-447

If the socket structure already points to a PCB, EINVAL is returned. in\_pcalloc allocates a new PCB, adds it to the front of UDP's PCB list, and links the socket structure and the PCB to each other.

448-450

soreserve reserves buffer space for a receive buffer and a send buffer for the socket. As noted in [Figure 16.7](#), soreserve just enforces system limits; the buffer

space is not actually allocated. The default values for the send and receive buffer sizes are 9216 bytes (`udp_sendspace`) and 41,600 bytes (`udp_recvspace`). The former allows for a maximum UDP datagram size of 9200 bytes (to hold 8 Kbytes of data in an NFS packet), plus the 16-byte `sockaddr_in` structure for the destination address. The latter allows for 40 1024-byte datagrams to be queued at one time for the socket. The process can change these defaults by calling `setsockopt`.

#### 451-452

There are two fields in the prototype IP header in the PCB that the process can change by calling `setsockopt`: the TTL and the TOS. The TTL defaults to 64 (`ip_defttl`) and the TOS defaults to 0 (normal service), since the PCB is initialized to 0 by `in_pcalloc`.

#### 453-455

The close system call issues the `PRU_DETACH` request. The function `udp_detach`, shown in [Figure 23.34](#), is

called. This function is also called later in this section for the PRU\_ABORT request.

### Figure 23.34. `udp_detach` function: delete a UDP PCB.

```
534 static void  
535 udp_detach(inp)  
536 struct inpcb *inp;  
537 {  
538     int     s = splnet();  
539     if (inp == udp_last_inpcb)  
540         udp_last_inpcb = &udb;  
541     in_pcbdetach(inp);  
542     splx(s);  
543 }
```

If the last-received PCB pointer (the one-behind cache) points to the PCB being detached, the cache pointer is set to the head of the UDP list (udb). The function `in_pcbdetach` removes the PCB from UDP's list and releases the PCB.

Returning to `udp_usrreq`, a PRU\_BIND request is the result of the bind system call and a PRU\_LISTEN request is the result of the listen system call. Both are shown in Figure 23.35.

### Figure 23.35. `udp_usrreq` function:

## **PRU\_BIND and PRU\_LISTEN requests.**

```
456     case PRU_BIND:  
457         s = splnet();  
458         error = in_pcbbind(inp, addr);  
459         splx(s);  
460         break;  
  
461     case PRU_LISTEN:  
462         error = EOPNOTSUPP;  
463         break;
```

*udp\_usrreq.c*

**456-460**

All the work for a PRU\_BIND request is done by in\_pcbbind.

**461-463**

The PRU\_LISTEN request is invalid for a connectionless protocol it is used only by connection-oriented protocols.

We mentioned earlier that a UDP application, either a client or server (normally a client), can call connect. This fixes the foreign IP address and port number that this socket can send to or receive from. [Figure 23.36](#) shows the PRU\_CONNECT, PRU\_CONNECT2, and PRU\_ACCEPT requests.

## Figure 23.36. udp\_usrreq function: PRU\_CONNECT, PRU\_CONNECT2, and PRU\_ACCEPT requests.

```
464     case PRU_CONNECT:                               udp_usrreq.c
465         if (inp->inp_faddr.s_addr != INADDR_ANY) {
466             error = EISCONN;
467             break;
468
469             s = splnet();
470             error = in_pcbconnect(inp, addr);
471             splx(s);
472             if (error == 0)
473                 soisconnected(so);
474             break;
475
476     case PRU_CONNECT2:
477         error = EOPNOTSUPP;
478         break;
479
480     case PRU_ACCEPT:
481         error = EOPNOTSUPP;
482         break;
```

464-474

If the socket is already connected, EISCONN is returned. The socket should never be connected at this point, because a call to connect on an already-connected UDP socket generates a PRU\_DISCONNECT request before this PRU\_CONNECT request. Otherwise in\_pcbconnect does all the work. If no errors are encountered, soisconnected marks the socket structure as being connected.

475-477

The socketpair system call issues the PRU\_CONNECT2 request, which is defined only for the Unix domain protocols.

478-480

The PRU\_ACCEPT request is from the accept system call, which is defined only for connection-oriented protocols.

The PRU\_DISCONNECT request can occur in two cases for a UDP socket:

**1. When a connected UDP socket is closed, PRU\_DISCONNECT is called before PRU\_DETACH.**

- When a connect is issued on an already-connected UDP socket, soconnect issues the PRU\_DISCONNECT request before the PRU\_CONNECT request.

[Figure 23.37](#) shows the PRU\_DISCONNECT request.

**Figure 23.37. udp\_usrreq function:  
PRU\_DISCONNECT request.**

```

481     case PRU_DISCONNECT:
482         if (inp->inp_faddr.s_addr == INADDR_ANY) {
483             error = ENOTCONN;
484             break;
485         }
486         s = splnet();
487         in_pcbservice(inp);
488         inp->inp_laddr.s_addr = INADDR_ANY;
489         splx(s);
490         so->so_state &= ~SS_ISCONNECTED; /* XXX */
491         break;

```

*udp\_usrreq.c*

If the socket is not already connected, ENOTCONN is returned. Otherwise `in_pcbservice` sets the foreign IP address to 0.0.0.0 and the foreign port to 0. The local address is also set to 0.0.0.0, since this PCB variable could have been set by `connect`.

A call to `shutdown` specifying that the process has finished sending data generates the PRU\_SHUTDOWN request, although it is rare for a process to issue this system call for a UDP socket. [Figure 23.38](#) shows the PRU\_SHUTDOWN, PRU\_SEND, and PRU\_ABORT requests.

**Figure 23.38. `udp_usrreq` function: PRU\_SHUTDOWN, PRU\_SEND, and PRU\_ABORT requests.**

```
492     case PRU_SHUTDOWN:
493         socantsendmore(so);
494         break;
495     case PRU_SEND:
496         return (udp_output(inp, m, addr, control));
497     case PRU_ABORT:
498         soisdisconnected(so);
499         udp_detach(inp);
500         break;
```

————— *udp\_usrreq.c*

## 492-494

socantsendmore sets the socket's flags to prevent any future output.

## 495-496

In [Figure 23.14](#) we showed how the five write functions ended up calling `udp_usrreq` with a `PRU_SEND` request. `udp_output` sends the datagram. `udp_usrreq` returns, to avoid falling through to the label `release` ([Figure 23.32](#)), since the mbuf chain containing the data (`m`) must not be released yet. IP output appends this mbuf chain to the appropriate interface output queue, and the device driver will release the mbuf when the data has been transmitted.

The only buffering of UDP output within the kernel is on the interface's output queue. If there is room in the socket's

send buffer for the datagram and destination address, sosend calls udp\_usrreq, which we see calls udp\_output. We saw in [Figure 23.20](#) that ip\_output is then called, which calls ether\_output for an Ethernet, placing the datagram onto the interface's output queue (if there is room). If the process calls sendto faster than the interface can transmit the datagrams, ether\_output can return ENOBUFS, which is returned to the process.

497-500

A PRU\_ABORT request should never be generated for a UDP socket, but if it is, the socket is disconnected and the PCB detached.

The PRU\_SOCKADDR and PRU\_PEERADDR requests are from the getsockname and getpeername system calls, respectively. These two requests, and the PRU\_SENSE request, are shown in [Figure 23.39](#).

**Figure 23.39. udp\_usrreq function:**

## PRU\_SOCKADDR, PRU\_PEERADDR, and PRU\_SENSE requests.

```
501     case PRU_SOCKADDR:
502         in_setsockaddr(inp, addr);
503         break;
504
505     case PRU_PEERADDR:
506         in_setpeeraddr(inp, addr);
507         break;
508
509     case PRU_SENSE:
510     /*
511      * fstat: don't bother with a blocksize.
512      */
513     return (0);
```

— udp\_usrreq.c

**501-506**

The functions `in_setsockaddr` and `in_setpeeraddr` fetch the information from the PCB, storing the result in the `addr` argument.

**507-511**

The `fstat` system call generates the `PRU_SENSE` request. The function returns OK, but doesn't return any other information. We'll see later that TCP returns the size of the send buffer as the `st_blksize` element of the stat structure.

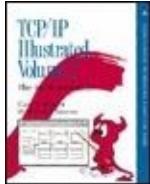
The remaining seven `PRU_xxx` requests, shown in [Figure 23.40](#), are not supported

for a UDP socket.

## Figure 23.40. `udp_usrreq` function: unsupported requests.

```
----- udp_usrreq.c
512     case PRU_SENDOOB:
513     case PRU_FASTTIMO:
514     case PRU_SLOWTIMO:
515     case PRU_PROTORCV:
516     case PRU_PROTOSEND:
517         error = EOPNOTSUPP;
518         break;
519     case PRU_RCVD:
520     case PRU_RCVOOB:
521         return (EOPNOTSUPP); /* do not free mbuf's */
----- udp_usrreq.c
```

There is a slight difference in how the last two are handled because PRU\_RCVD doesn't pass a pointer to an mbuf as an argument (m is a null pointer) and PRU\_RCVOOB passes a pointer to an mbuf for the protocol to fill in. In both cases the error is immediately returned, without breaking out of the switch and releasing the mbuf chain. With PRU\_RCVOOB the caller releases the mbuf that it allocated.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.11 udp\_sysctl Function

The sysctl function for UDP supports only a single option, the UDP checksum flag. The system administrator can enable or disable UDP checksums using the sysctl(8) program. [Figure 23.41](#) shows the udp\_sysctl function. This function calls sysctl\_int to fetch or set the value of the integer udpcksum.

**Figure 23.41. udp\_sysctl function.**

```
547 udp_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
548 int      *name;
549 u_int     namelen;
550 void     *oldp;
551 size_t   *oldlenp;
552 void     *newp;
553 size_t   newlen;
554 {
555     /* All sysctl names at this level are terminal. */
556     if (namelen != 1)
557         return (ENOTDIR);
558     switch (name[0]) {
559     case UDPCTL_CHECKSUM:
560         return (sysctl_int(oldp, oldlenp, newp, newlen, &udpcksum));
561     default:
562         return (ENOPROTOOPT);
563     }
564     /* NOTREACHED */
565 }
```

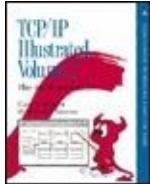
*udp\_usrreq.c*

---

**Team-Fly** 

[◀ Previous](#) [Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.12 Implementation Refinements

#### UDP PCB Cache

In [Section 22.12](#) we talked about some general features of PCB searching and how the code we've seen uses a linear search of the protocol's PCB list. We now tie this together with the one-behind cache used by UDP in [Figure 23.24](#).

The problem with the one-behind cache occurs when the cached PCB contains wildcard values (for either the local address, foreign address, or foreign port):

the cached value never matches any received datagram. One solution tested in [Partridge and Pink 1993] is to modify the cache to not compare wildcarded values. That is, instead of comparing the foreign address in the PCB with the source address in the datagram, compare these two values only if the foreign address in the PCB is not a wildcard.

There's a subtle problem with this approach [Partridge and Pink 1993]. Assume there are two sockets bound to local port 555. One has the remaining three elements wildcarded, while the other has connected to the foreign address 128.1.2.3 and the foreign port 1600. If we cache the first PCB and a datagram arrives from 128.1.2.3, port 1600, we can't ignore comparing the foreign addresses just because the cached value has a wildcarded foreign address. This is called *cache hiding*. The cached PCB has hidden another PCB that is a better match in this example.

To get around cache hiding requires more work when a new entry is added to or

deleted from the cache. Those PCBs that hide other PCBs cannot be cached. This is not a problem, however, because the normal scenario is to have one socket per local port. The example we just gave with two sockets bound to local port 555, while possible (especially on a multihomed host), is rare.

The next enhancement tested in [Partridge and Pink 1993] is to also remember the PCB of the last datagram sent. This is motivated by [Mogul 1991], who shows that half of all datagrams received are replies to the last datagram that was sent. Cache hiding is a problem here also, so PCBs that would hide other PCBs are not cached.

The results of these two caches shown in [Partridge and Pink 1993] on a general-purpose system measured for around 100,000 received UDP datagrams show a 57% hit rate for the last-received PCB cache and a 30% hit rate for the last-sent PCB cache. The amount of CPU time spent in `udp_input` is more than halved, compared to the version with no caching.

These two caches still depend on a certain amount of locality: that with a high probability the UDP datagram that just arrived is either from the same peer as the last UDP datagram received or from the peer to whom the last datagram was sent. The latter is typical for request-response applications that send a datagram and wait for a reply. [McKenney and Dove 1992] show that some applications, such as data entry into an online transaction processing (OLTP) system, don't yield the high cache hit rates that [Partridge and Pink 1993] observed. As we mentioned in [Section 22.12](#), placing the PCBs onto hash chains provided an order of magnitude improvement over the last-received and last-sent caches for a system with thousands of OLTP connections.

## UDP Checksum

The next area for improving the implementation is to combine the copying of data between the process and the kernel with the calculation of the checksum. In Net/3, each byte of data is

processed twice during an output operation: once when copied from the process into an mbuf (the function uiomove, which is called by sosend), and again when the UDP checksum is calculated (by the function in\_cksum, which is called by udp\_output). This happens on input as well as output.

[Partridge and Pink 1993] modified the UDP output processing from what we showed in [Figure 23.14](#) so that a UDP-specific function named udp\_sosend is called instead of sosend. This new function calculates the checksum of the UDP header and the pseudo-header in-line (instead of calling the general-purpose function in\_cksum) and then copies the data from the process into an mbuf chain using a special function named in\_uiomove (instead of the general-purpose uiomove). This new function copies the data *and* updates the checksum. The amount of time spent copying the data and calculating the checksum is reduced with this technique by about 40 to 45%.

On the receive side the scenario is

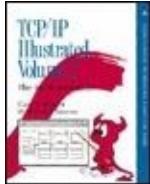
different. UDP calculates the checksum of the UDP header and the pseudo-header, removes the UDP header, and queues the data for the appropriate socket. When the application reads the data, a special version of soreceive (called udp\_soreceive) completes the calculation of the checksum while copying the data into the user's buffer. If the checksum is in error, however, the error is not detected until the entire datagram has been copied into the user's buffer. In the normal case of a blocking socket, udp\_soreceive just waits for the next datagram to arrive. But if the socket is nonblocking, the error EWOULDBLOCK must be returned if another datagram is not ready to be passed to the process. This implies two changes in the socket interface for a nonblocking read from a UDP socket:

- 1. The select function can indicate that a nonblocking UDP socket is readable, yet the error EWOULDBLOCK is unexpectedly returned by one of the read functions if the checksum fails.**

- Since a checksum error is detected after the datagram has been copied into the user's buffer, the application's buffer is changed even though no data is returned by the read.

Even with a blocking socket, if the datagram with the checksum error contains 100 bytes of data and the next datagram without an error contains 40 bytes of data, recvfrom returns a length of 40, but the 60 bytes that follow in the user's buffer have also been modified.

[[Partridge and Pink 1993](#)] compare the timings for a copy versus a copy-with-checksum for six different computers. They show that the checksum is calculated for free during the copy operation on many architectures. This occurs when memory access speeds and CPU processing speeds are mismatched, as is true for many current RISC processors.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 23. UDP: User Datagram Protocol

### 23.13 Summary

UDP is a simple, connectionless protocol, which is why we cover it before looking at TCP. UDP output is simple: IP and UDP headers are prepended to the user's data, as much of the header is filled in as possible, and the result is passed to `ip_output`. The only complication is calculating the UDP checksum, which involves prepending a pseudo-header just for the checksum computation. We'll encounter a similar pseudo-header for the calculation of the TCP checksum in [Chapter 26](#).

When `udp_input` receives a datagram, it

first performs a general validation (the length and checksum); the processing then differs depending on whether the destination IP address is a unicast address or a broadcast or multicast address. A unicast datagram is delivered to at most one process, but a broadcast or multicast datagram can be delivered to multiple processes. A one-behind cache is maintained for unicast datagrams, which maintains a pointer to the last Internet PCB for which a UDP datagram was received. We saw, however, that because of the prevalence of wildcard addressing with UDP applications, this cache is practically useless.

The `udp_ctlinput` function is called to handle received ICMP messages, and the `udp_usrreq` function handles the `PRU_XXX` requests from the socket layer.

## Exercises

List the five types of mbuf chains  
**23.1** that `udp_output` passes to

`ip_output`. (*Hint:* look at `sosend`.)

What happens to the answer for the  
**23.2** previous exercise when the process specifies IP options for the outgoing datagram?

**23.3** Does a UDP client need to call bind?  
Why or why not?

What happens to the processor priority level in `udp_output` if the  
**23.4** socket is unconnected and the call to `M_PREPEND` in Figure 23.15 fails?

**23.5** `udp_output` does not check for a destination port of 0. Is it possible to send a UDP datagram with a destination port of 0?

Assuming the `IP_RECVDSTADDR` socket option worked when a datagram was sent to a broadcast

**23.6** address, how can you then determine if this address is a broadcast address?

Who releases the mbuf that  
**23.7** `udp_saveopt` ([Figure 23.28](#)) allocates?

How can a process disconnect a connected UDP socket? That is, the process calls connect and exchanges datagrams with that peer, and then the process wants to disconnect the socket, allowing it to call sendto and send a datagram to some other host.

In our discussion of [Figure 22.25](#) we noted that a UDP application that calls connect with a foreign IP address of 255.255.255.255 actually sends datagrams out the primary interface with a destination **23.9** IP address corresponding to the

broadcast address of that interface. What happens if a UDP application uses an unconnected socket instead, calling `sendto` with a destination address of 255.255.255.255?

**23.10**

After discussing the problem with [Figure 23.27](#), we mentioned that this problem would not exist if the server used the destination IP address from the request as the source IP address of the reply. Explain how the server could do this.

**23.11**

Implement changes to allow a process to perform path MTU discovery using UDP: the process must be able to set the "don't fragment" bit in the resulting IP datagram and be told if the corresponding ICMP destination unreachable error is received.

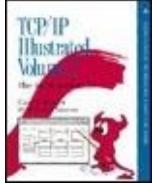
**23.12** Does the variable `udp_in` need to be global?

Modify `udp_input` to save the IP options and make them available to the receiver with the `IP_RECVOPTS` socket option.

**23.14** Fix the one-behind cache in Figure 23.24.

Fix `udp_input` to implement the `IP_RECVOPTS` and `IP_RTOPTS` socket options.

**23.16** Fix `udp_input` so that the `IP_RECVDSTADDR` socket option works for datagrams sent to a broadcast or multicast address.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 24. TCP: Transmission Control Protocol

[Section 24.1. Introduction](#)

[Section 24.2. Code Introduction](#)

[Section 24.3. TCP protosw Structure](#)

[Section 24.4. TCP Header](#)

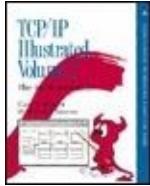
[Section 24.5. TCP Control Block](#)

[Section 24.6. TCP State Transition Diagram](#)

[Section 24.7. TCP Sequence Numbers](#)

[Section 24.8. tcp\\_init Function](#)

[Section 24.9. Summary](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.1 Introduction

The Transmission Control Protocol, or TCP, provides a connection-oriented, reliable, byte-stream service between the two end points of an application. This is completely different from UDP's connectionless, unreliable, datagram service.

The implementation of UDP presented in [Chapter 23](#) comprised 9 functions and about 800 lines of C code. The TCP implementation we're about to describe comprises 28 functions and almost 4,500 lines of C code. Therefore we divide the presentation of TCP into multiple chapters.

These chapters are not an introduction to TCP. We assume the reader is familiar with the operation of TCP from Chapters 17-24 of Volume 1.

---

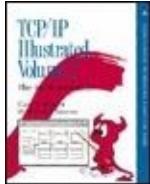
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.2 Code Introduction

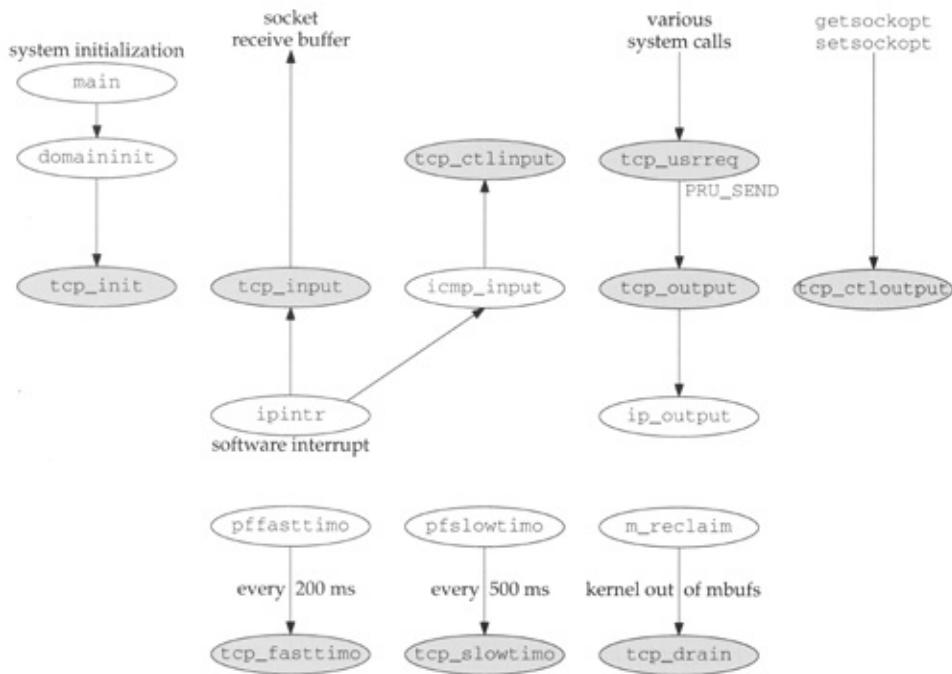
The TCP functions appear in six C files and numerous TCP definitions are in seven headers, as shown in [Figure 24.1](#).

**Figure 24.1. Files discussed in the TCP chapters.**

File	Description
netinet/tcp.h	tcp_hdr structure definition
netinet/tcp_debug.h	tcp_debug structure definition
netinet/tcp_fsm.h	definitions for TCP's finite state machine
netinet/tcp_seq.h	macros for comparing TCP sequence numbers
netinet/tcp_timer.h	definitions for TCP timers
netinet/tcp_var.h	tcpcb (control block) and tcpstat (statistics) structure definitions
netinet/tcpip.h	TCP plus IP header definition
netinet/tcp_debug.c	support for SO_DEBUG socket debugging (Section 27.10)
netinet/tcp_input.c	tcp_input and ancillary functions (Chapters 28 and 29)
netinet/tcp_output.c	tcp_output and ancillary functions (Chapter 26)
netinet/tcp_subr.c	miscellaneous TCP subroutines (Chapter 27)
netinet/tcp_timer.c	TCP timer handling (Chapter 25)
netinet/tcp_usrreq.c	PRU_XXX request handling (Chapter 30)

[Figure 24.2](#) shows the relationship of the various TCP functions to other kernel functions. The shaded ellipses are the nine main TCP functions that we cover. Eight of these functions appear in the TCP protosw structure ([Figure 24.8](#)) and the ninth is `tcp_output`.

## Figure 24.2. Relationship of TCP functions to rest of the kernel.



## Global Variables

[Figure 24.3](#) shows the global variables we

encounter throughout the TCP functions.

### Figure 24.3. Global variables introduced in the following chapters.

Variable	Datatype	Description
tcb	struct inpcb	head of the TCP Internet PCB list
tcp_last_inpcb	struct inpcb *	pointer to PCB for last received segment: one-behind cache
tcpstat	struct tcpstat	TCP statistics (Figure 24.4)
tcp_outflags	u_char	array of output flags, indexed by connection state (Figure 24.16)
tcp_recvspace	u_long	default size of socket receive buffer (8192 bytes)
tcp_sendspace	u_long	default size of socket send buffer (8192 bytes)
tcp_iss	tcp_seq	initial send sequence number (ISS)
tcp_rexmtthresh	int	number of duplicate ACKs to trigger fast retransmit (3)
tcp_mssdfilt	int	default MSS (512 bytes)
tcp_rttdfilt	int	default RTT if no data (3 seconds)
tcp_do_rfc1323	int	if true (default), request window scale and timestamp options
tcp_now	u_long	500 ms counter for RFC 1323 timestamps
tcp_keepidle	int	keepalive: idle time before first probe (2 hours)
tcp_keepintvl	int	keepalive: interval between probes when no response (75 sec) (also used as timeout for connect)
tcp_maxidle	int	keepalive: time after probing before giving up (10 min)

## Statistics

Various TCP statistics are maintained in the global structure `tcpstat`, described in [Figure 24.4](#). We'll see where these counters are incremented as we proceed through the code.

### Figure 24.4. TCP statistics maintained in the `tcpstat` structure.

tcpstat member	Description	Used by SNMP
tcps_accepts	#SYNs received in LISTEN state	•
tcps_closed	#connections closed (includes drops)	•
tcps_connattempt	#connections initiated (calls to connect)	•
tcps_connndrops	#embryonic connections dropped (before SYN received)	•
tcps_connects	#connections established actively or passively	•
tcps_delack	#delayed ACKs sent	•
tcps_drops	#connections dropped (after SYN received)	•
tcps_keepdrops	#connections dropped in keepalive (established or awaiting SYN)	•
tcps_keepprobe	#keepalive probes sent	•
tcps_keeptimeo	#times keepalive timer or connection-establishment timer expire	•
tcps_pawsdrop	#segments dropped due to PAWS	•
tcps_pcbschemmiss	#times PCB cache comparison fails	•
tcps_persistimeo	#times persist timer expires	•
tcps_predack	#times header prediction correct for ACKs	•
tcps_preddat	#times header prediction correct for data packets	•
tcps_rcvackbyte	#bytes ACKed by received ACKs	•
tcps_rcvackpack	#received ACK packets	•
tcps_rcvacktoomuch	#received ACKs for unsent data	•
tcps_rcvafterclose	#packets received after connection closed	•
tcps_rcvbadoff	#packets received with invalid header length	•
tcps_rcvbadsum	#packets received with checksum errors	•
tcps_rcvbyte	#bytes received in sequence	•
tcps_rcvbyteafterwin	#bytes received beyond advertised window	•
tcps_rcvdupack	#duplicate ACKs received	•
tcps_rcvdupbyte	#bytes received in completely duplicate packets	•
tcps_rcvduppack	#packets received with completely duplicate bytes	•
tcps_rcvoobyte	#out-of-order bytes received	•
tcps_rcvoopack	#out-of-order packets received	•
tcps_rcvpack	#packets received in sequence	•
tcps_rcvpackafterwin	#packets with some data beyond advertised window	•
tcps_rcvpardupbyte	#duplicate bytes in part-duplicate packets	•
tcps_rcvparduppack	#packets with some duplicate data	•
tcps_rcvshort	#packets received too short	•
tcps_rcvtotal	total #packets received	•
tcps_rcvwinprobe	#window probe packets received	•
tcps_rcvwinupd	#received window update packets	•
tcps_rexmtimeo	#retransmit timeouts	•
tcps_rttupdated	#times RTT estimators updated	•
tcps_segstimed	#segments for which TCP tried to measure RTT	•
tcps_sndacks	#ACK-only packets sent (data length = 0)	•
tcps_sndbyte	#data bytes sent	•
tcps_sndctrl	#control (SYN, FIN, RST) packets sent (data length = 0)	•
tcps_sndpack	#data packets sent (data length > 0)	•
tcps_sndprobe	#window probes sent (1 byte of data forced by persist timer)	•
tcps_ndrexmitbyte	#data bytes retransmitted	•
tcps_ndrexmitpack	#data packets retransmitted	•
tcps_ndtotal	total #packets sent	•
tcps_ndurg	#packets sent with URG-only (data length = 0)	•
tcps_ndwinup	#window update-only packets sent (data length = 0)	•
tcps_timeoutdrop	#connections dropped in retransmission timeout	•

Figure 24.5 shows some sample output of these statistics, from the netstat s command. These statistics were collected after the host had been up for 30 days. Since some counters come in pairs one counts the number of packets and the

other the number of bytes we abbreviate these in the figure. For example, the two counters for the second line of the table are tcps\_sndpack and tcps\_sndbyte.

## Figure 24.5. Sample TCP statistics.

netstat -s output	tcpstat members
10,655,999 packets sent 9,177,823 data packets (-22,194,928 bytes) 257,295 data packets (81,075,086 bytes) retransmitted 862,900 ack-only packets (531,285 delayed) 229 URG-only packets 3,453 window probe packets 74,925 window update packets 279,387 control packets	tcps_sndtotal tcps_snd{pack,byte} tcps_sndrexmit{pack,byte} tcps_sndacks,tcps_delack tcps_endurg tcps_sndprobe tcps_endwinup tcps_sndctrl
8,801,953 packets received 6,617,079 acks (for -21,264,360 bytes) 235,311 duplicate acks 0 acks for unsent data 4,670,615 packets (324,965,351 bytes) rcvd in-sequence 46,953 completely duplicate packets (1,549,785 bytes) 22 old duplicate packets 3,442 packets with some dup. data (54,483 bytes duped) 77,114 out-of-order packets (13,938,456 bytes) 1,892 packets (1,755 bytes) of data after window 1,755 window probes 175,476 window update packets 1,017 packets received after close 60,370 discarded for bad checksums 279 discarded for bad header offset fields 0 discarded because packet too short	tcps_rcvtotal tcps_revack{pack,byte} tcps_rcvdupack tcps_rcvacktoomuch tcps_rcv{pack,byte} tcps_rcvdup{pack,byte} tcps_pawsdrop tcps_rcvpartdup{pack,byte} tcps_revo{o}{pack,byte} tcps_rcv{pack,byte}afterwin tcps_rcvwindup tcps_rcvafterclose tcps_rcvbadsum tcps_rcvbadoff tcps_rcvshort
144,020 connection requests 92,595 connection accepts 126,820 connections established (including accepts) 237,743 connections closed (including 1,061 drops) 110,016 embryonic connections dropped	tcps_connattempt tcps_accepts tcps_connects tcps_closed,tcps_drops tcps_conndrops
6,363,546 segments updated rtt (of 6,444,667 attempts) 114,797 retransmit timeouts 86 connection dropped by rexmit timeout 1,173 persist timeouts 16,419 keepalive timeouts 6,899 keepalive probes sent 3,219 connections dropped by keepalive	tcps_{rttupdated,segstimed} tcps_rexmttimeo tcps_timeoutdrop tcps_persisttimeo tcps_keeptimeo tcps_kepprobe tcps_kepdrops
733,130 correct ACK header predictions 1,266,889 correct data packet header predictions 1,851,557 cache misses	tcps_predack tcps_preddat tcps_pcbaclmiss

The counter for tcps\_sndbyte should be 3,722,884,824, not -22,194,928 bytes. This is an average of about 405 bytes

per segment, which makes sense. Similarly, the counter for tcps\_rcvackbyte should be 3,738,811,552, not -21,264,360 bytes (for an average of about 565 bytes per segment). These numbers are incorrectly printed as negative numbers because the printf calls in the netstat program use %d (signed decimal) instead of %lu (long integer, unsigned decimal). All the counters are unsigned long integers, and these two counters are near the maximum value of an unsigned 32-bit long integer ( $2^{32} = 4,294,967,295$ ).

## SNMP Variables

Figure 24.6 shows the 14 simple SNMP variables in the TCP group and the counters from the tcpstat structure implementing that variable. The constant values shown for the first four entries are fixed by the Net/3 implementation. The counter tcpCurrEstab is computed as the number of Internet PCBs on the TCP PCB list.

## Figure 24.6. Simple SNMP variables in tcp group.

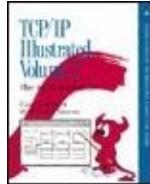
SNMP variable	tcpstat members or constant	Description
tcpRtoAlgorithm	4	algorithm used to calculate retransmission timeout value: 1 = none of the following, 2 = a constant RTO, 3 = MIL-STD-1778 Appendix B, 4 = Van Jacobson's algorithm.
tcpRtoMin	1000	minimum retransmission timeout value, in milliseconds
tcpRtoMax	64000	maximum retransmission timeout value, in milliseconds
tcpMaxConn	-1	maximum #TCP connections (-1 if dynamic)
tcpActiveOpens	tcps_connattempt	#transitions from CLOSED to SYN_SENT states
tcpPassiveOpens	tcps_accepts	#transitions from LISTEN to SYN_RCVD states
tcpAttemptFails	tcps_conndrops	#transitions from SYN_SENT or SYN_RCVD to CLOSED, plus #transitions from SYN_RCVD to LISTEN
tcpEstabResets	tcps_drops	#transitions from ESTABLISHED or CLOSE_WAIT states to CLOSED
tcpCurrEstab	(see text)	#connections currently in ESTABLISHED or CLOSE_WAIT states
tcpInSegs	tcps_rcvtotal	total #segments received
tcpOutSegs	tcps_sndtotal - tcps_endrexmitpack	total #segments sent, excluding those containing only retransmitted bytes
tcpRetransSegs	tcps_endrexmitpack	total #retransmitted segments
tcpInErrs	tcps_rcvbadsum + tcps_rcvbadoff + tcps_rcvshort	total #segments received with an error
tcpOutRsts	(not implemented)	total #segments sent with RST flag set

Figure 24.7 shows `tcpTable`, the TCP listener table.

## Figure 24.7. Variables in TCP listener table: `tcpTable`.

index = <tcpConnLocalAddress>,<tcpConnLocalPort>,<tcpConnRemAddress>,<tcpConnRemPort>		
SNMP variable	PCB variable	Description
tcpConnState	t_state	state of connection: 1 = CLOSED, 2 = LISTEN, 3 = SYN_SENT, 4 = SYN_RCVD, 5 = ESTABLISHED, 6 = FIN_WAIT_1, 7 = FIN_WAIT_2, 8 = CLOSE_WAIT, 9 = LAST_ACK, 10 = CLOSING, 11 = TIME_WAIT, 12 = delete TCP control block.
tcpConnLocalAddress	inp_laddr	local IP address
tcpConnLocalPort	inp_lport	local port number
tcpConnRemAddress	inp_faddr	foreign IP address
tcpConnRemPort	inp_fport	foreign port number

The first PCB variable (`t_state`) is from the TCP control block ([Figure 24.13](#)) and the remaining four are from the Internet PCB ([Figure 22.4](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.3 TCP protosw Structure

Figure 24.8 lists the TCP protosw structure, the protocol switch entry for TCP.

#### Figure 24.8. The TCP protosw structure.

Member	inetsw[2]	Description
pr_type	<i>SOCK_STREAM</i>	TCP provides a byte-stream service
pr_domain	<i>&amp;inetdomain</i>	TCP is part of the Internet domain
pr_protocol	<i>IPPROTO_TCP (6)</i>	appears in the <i>ip_p</i> field of the IP header
pr_flags	<i>PR_CONNREQUIRED / PR_WANTRCV</i>	socket layer flags, not used by protocol processing
pr_input	<i>tcp_input</i>	receives messages from IP layer
pr_output	<i>0</i>	not used by TCP
pr_ctlinput	<i>tcp_ctlinput</i>	control input function for ICMP errors
pr_ctloutput	<i>tcp_ctloutput</i>	respond to administrative requests from a process
pr_usrreq	<i>tcp_usrreq</i>	respond to communication requests from a process
pr_init	<i>tcp_init</i>	initialization for TCP
pr_fasttimo	<i>tcp_fasttimo</i>	fast timeout function, called every 200 ms
pr_slowtimo	<i>tcp_slowtimo</i>	slow timeout function, called every 500 ms
pr_drain	<i>tcp_drain</i>	called when kernel runs out of mbufs
pr_sysctl	<i>0</i>	not used by TCP

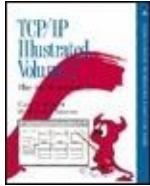
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.4 TCP Header

The TCP header is defined as a `tcp_hdr` structure. [Figure 24.9](#) shows the C structure and [Figure 24.10](#) shows a picture of the TCP header.

**Figure 24.9. `tcp_hdr` structure.**

---

```

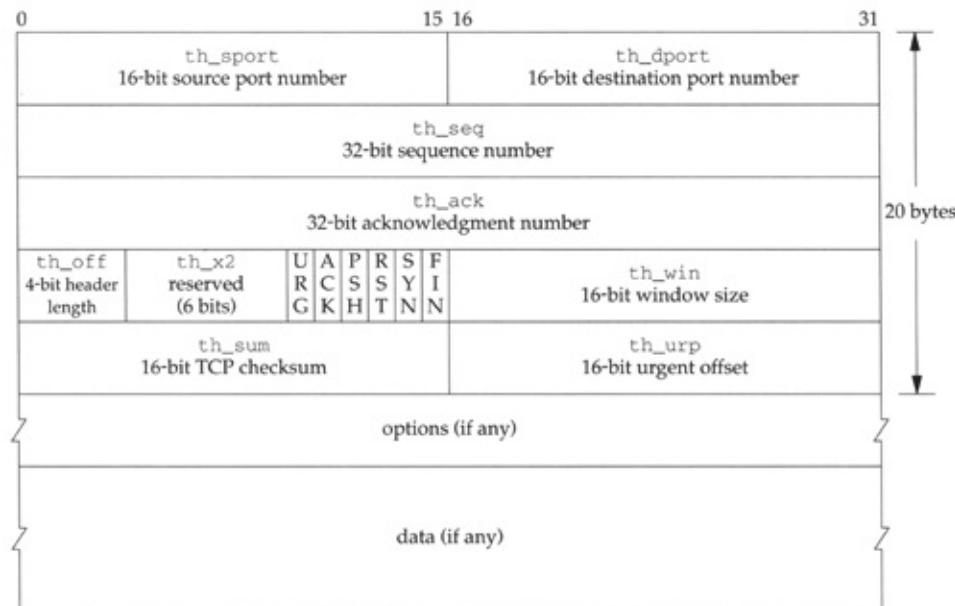
40 struct tcphdr {
41     u_short th_sport;           /* source port */
42     u_short th_dport;           /* destination port */
43     tcp_seq th_seq;            /* sequence number */
44     tcp_seq th_ack;            /* acknowledgement number */
45 #if BYTE_ORDER == LITTLE_ENDIAN
46     u_char th_x2:4;           /* (unused) */
47     th_off:4;                 /* data offset */
48#endif
49 #if BYTE_ORDER == BIG_ENDIAN
50     u_char th_off:4;           /* data offset */
51     th_x2:4;                 /* (unused) */
52#endif
53     u_char th_flags;           /* ACK, FIN, PUSH, RST, SYN, URG */
54     u_short th_win;            /* advertised window */
55     u_short th_sum;            /* checksum */
56     u_short th_urp;            /* urgent offset */
57 };

```

---

tcp.h

**Figure 24.10. TCP header and optional data.**



Most RFCs, most books (including Volume 1), and the code we'll examine call `th_urp` the *urgent pointer*. A better term is the *urgent offset*, since this field

is a 16-bit unsigned offset that must be added to the sequence number field (`th_seq`) to give the 32-bit sequence number of the *last* byte of urgent data. (There is a continuing debate over whether this sequence number points to the last byte of urgent data or to the byte that follows. This is immaterial for the present discussion.) We'll see in [Figure 24.13](#) that TCP correctly calls the 32-bit sequence number of the last byte of urgent data `snd_up` the *send urgent pointer*. But using the term *pointer* for the 16-bit offset in the TCP header is misleading. In [Exercise 26.6](#) we'll reiterate the distinction between the urgent pointer and the urgent offset.

The 4-bit header length, the 6 reserved bits that follow, and the 6 flag bits are defined in C as two 4-bit bit-fields, followed by 8 bits of flags. To handle the difference in the order of these 4-bit fields within an 8-bit byte, the code contains an `#ifdef` based on the byte order of the system.

Also notice that we call the 4-bit `th_off` the

*header length*, while the C code calls it the *data offset*. Both are correct since it is the length of the TCP header, including options, in 32-bit words, which is the offset of the first byte of data.

The `th_flags` member contains 6 flag bits, accessed using the names in [Figure 24.11](#).

**Figure 24.11. `th_flags` values.**

<code>th_flags</code>	Description
<code>TH_ACK</code>	the acknowledgment number ( <code>th_ack</code> ) is valid
<code>TH_FIN</code>	the sender is finished sending data
<code>TH_PUSH</code>	receiver should pass the data to application without delay
<code>TH_RST</code>	reset the connection
<code>TH_SYN</code>	synchronize sequence numbers (establish connection)
<code>TH_URG</code>	the urgent offset ( <code>th_urp</code> ) is valid

In Net/3 the TCP header is normally referenced as an IP header immediately followed by a TCP header. This is how `tcp_input` processes received IP datagrams and how `tcp_output` builds outgoing IP datagrams. This combined IP/TCP header is a `tcpiphdr` structure, shown in [Figure 24.12](#).

**Figure 24.12. `tcpiphdr` structure:**

## combined IP/TCP header.

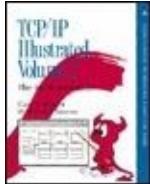
```
38 struct tcphdr {  
39     struct ipovly ti_i;           /* overlaid ip structure */  
40     struct tchdr ti_t;          /* tcp header */  
41 };  
  
42 #define ti_next    ti_i.ih_next  
43 #define ti_prev    ti_i.ih_prev  
44 #define ti_xl      ti_i.ih_xl  
45 #define ti_pr      ti_i.ih_pr  
46 #define ti_len     ti_i.ih_len  
47 #define ti_src     ti_i.ih_src  
48 #define ti_dst     ti_i.ih_dst  
49 #define ti_sport   ti_t.th_sport  
50 #define ti_dport   ti_t.th_dport  
51 #define ti_seq     ti_t.th_seq  
52 #define ti_ack     ti_t.th_ack  
53 #define ti_x2     ti_t.th_x2  
54 #define ti_off     ti_t.th_off  
55 #define ti_flags   ti_t.th_flags  
56 #define ti_win     ti_t.th_win  
57 #define ti_sum     ti_t.th_sum  
58 #define ti_urp     ti_t.th_urp
```

tcpip.h

tcpip.h

### 38-58

The 20-byte IP header is defined as an ipovly structure, which we showed earlier in [Figure 23.12](#). As we discussed with [Figure 23.19](#), this structure is not a real IP header, although the lengths are the same (20 bytes).



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.5 TCP Control Block

In [Figure 22.1](#) we showed that TCP maintains its own control block, a `tcpcb` structure, in addition to the standard Internet PCB. In contrast, UDP has everything it needs in the Internet PCB; it doesn't need its own control block.

The TCP control block is a large structure, occupying 140 bytes. As shown in [Figure 22.1](#) there is a one-to-one relationship between the Internet PCB and the TCP control block, and each points to the other. [Figure 24.13](#) shows the definition of the TCP control block.

## Figure 24.13. tcpcb structure: TCP control block.

```

tcp_var.h
41 struct tcpcb {
42     struct tciphdr *seg_next; /* reassembly queue of received segments */
43     struct tciphdr *seg_prev; /* reassembly queue of received segments */
44     short t_state;           /* connection state (Figure 24.16) */
45     short t_timer[TCPT_NTIMERS]; /* tcp timers (Chapter 25) */
46     short t_rxtshift;        /* log(2) of rexmt exp. backoff */
47     short t_rxtcur;          /* current retransmission timeout (#ticks) */
48     short t_dupacks;         /* #consecutive duplicate ACKs received */
49     u_short t_maxseg;        /* maximum segment size to send */
50     char t_force;            /* 1 if forcing out a byte (persist/OOB) */
51     u_short t_flags;          /* (Figure 24.14) */
52     struct tciphdr *t_template; /* skeletal packet for transmit */
53     struct inpcb *t_inpcb;    /* back pointer to internet PCB */
54 /*
55  * The following fields are used as in the protocol specification.
56  * See RFC783, Dec. 1981, page 21.
57 */
58 /* send sequence variables */
59     tcp_seq snd_una;          /* send unacknowledged */
60     tcp_seq snd_nxt;          /* send next */
61     tcp_seq snd_up;           /* send urgent pointer */
62     tcp_seq snd_w11;          /* window update seg seq number */
63     tcp_seq snd_w12;          /* window update seg ack number */
64     tcp_seq ihs;               /* initial send sequence number */
65     u_long snd_wnd;           /* send window */
66 /* receive sequence variables */
67     u_long rcv_wnd;           /* receive window */
68     tcp_seq rcv_nxt;          /* receive next */
69     tcp_seq rcv_up;           /* receive urgent pointer */
70     tcp_seq irs;               /* initial receive sequence number */
71 /*
72  * Additional variables for this implementation.
73 */
74 /* receive variables */
75     tcp_seq rcv_adv;           /* advertised window by other end */
76 /* retransmit variables */
77     tcp_seq snd_max;           /* highest sequence number sent;
78                                * used to recognize retransmits */
79 /* congestion control (slow start, source quench, retransmit after loss) */
80     u_long snd_cwnd;           /* congestion-controlled window */
81     u_long snd_ssthresh;        /* snd_cwnd size threshhold for slow start
82                                * exponential to linear switch */
83 /*
84  * transmit timing stuff. See below for scale of srtt and rttvar.
85  * "Variance" is actually smoothed difference.
86 */
87     short t_idle;                /* inactivity time */
88     short t_rtt;                 /* round-trip time */
89     tcp_seq t_rtseq;             /* sequence number being timed */
90     short t_srtt;                /* smoothed round-trip time */
91     short t_rttvar;               /* variance in round-trip time */
92     u_short t_rttmin;              /* minimum rtt allowed */
93     u_long max_sndwnd;            /* largest window peer has offered */

```

```

94 /* out-of-band data */
95     char    t_oobflags;          /* TCPOOB_HAVEDATA, TCPOOB_HADDATA */
96     char    t_iobc;             /* input character, if not SO_OOBINLINE */
97     short   t_softerror;        /* possible error not yet reported */
98 /* RFC 1323 variables */
99     u_char   snd_scale;         /* scaling for send window (0-14) */
100    u_char   rcv_scale;         /* scaling for receive window (0-14) */
101    u_char   request_r_scale;   /* our pending window scale */
102    u_char   requested_s_scale; /* peer's pending window scale */
103    u_long   ts_recent;         /* timestamp echo data */
104    u_long   ts_recent_age;     /* when last updated */
105    tcp_seq  last_ack_sent;     /* sequence number of last ack field */
106 };
107 #define intotcpb(ip)  ((struct tcpcb *)(ip)->inp_ppcb)
108 #define sototcpb(so)  (intotcpb(sotoinpcb(so)))

```

*tcp\_var.h*

We'll save the discussion of these variables until we encounter them in the code.

Figure 24.14 shows the values for the `t_flags` member.

## Figure 24.14. `t_flags` values.

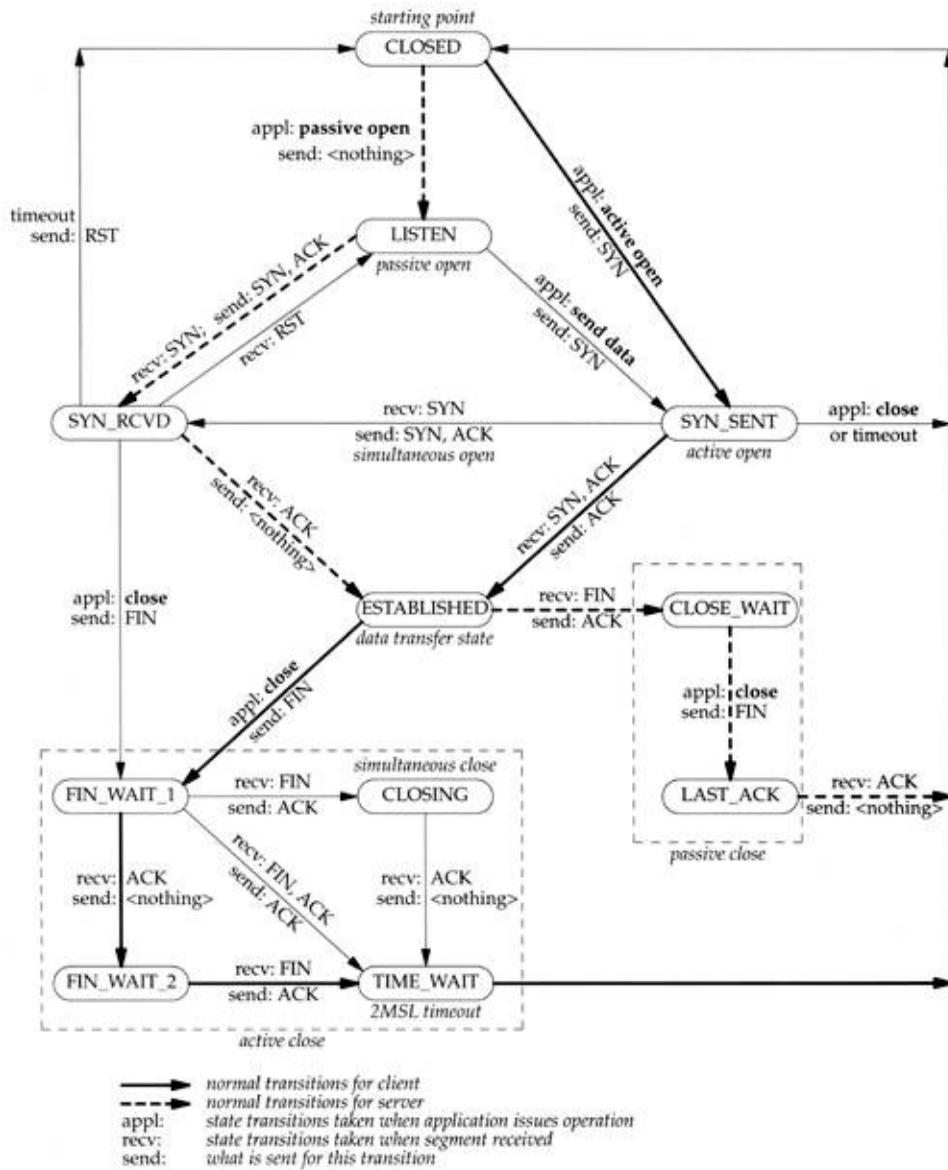
<code>t_flags</code>	Description
<code>TF_ACKNOW</code>	send ACK immediately
<code>TF_DELACK</code>	send ACK, but try to delay it
<code>TF_NODELAY</code>	don't delay packets to coalesce (disable Nagle algorithm)
<code>TF_NOOPT</code>	don't use TCP options (never set)
<code>TF_SENDFIN</code>	have sent FIN
<code>TF_RCVD_SCALE</code>	set when other side sends window scale option in SYN
<code>TF_RCVD_TSTMP</code>	set when other side sends timestamp option in SYN
<code>TF_REQ_SCALE</code>	have/will request window scale option in SYN
<code>TF_REQ_TSTMP</code>	have/will request timestamp option in SYN

## Chapter 24. TCP: Transmission Control

### 24.6 TCP State Transition Diagram

Many of TCP's actions, in response to different events, can be summarized in a state transition diagram. [Figure 24.15](#). We also duplicate this diagram on one of the TCP chapters for reference while reading the TCP chapters.

**Figure 24.15. TCP state transition diagram**



These state transitions define the TCP finite state transition from LISTEN to SYN\_SENT is allowed this using the sockets API (i.e., a connect is not

The t\_state member of the control block holds with the values shown in [Figure 24.16](#).

**Figure 24.16. t\_state**

t_state	value	Description	tcp_outflags[]
TCPS_CLOSED	0	closed	TH_RST / TH_ACK
TCPS_LISTEN	1	listening for connection (passive open)	0
TCPS_SYN_SENT	2	have sent SYN (active open)	TH_SYN
TCPS_SYN_RECEIVED	3	have sent and received SYN; awaiting ACK	TH_SYN / TH_ACK
TCPS_ESTABLISHED	4	established (data transfer)	TH_ACK
TCPS_CLOSE_WAIT	5	received FIN, waiting for application close	TH_ACK
TCPS_FIN_WAIT_1	6	have closed, sent FIN; awaiting ACK and FIN	TH_FIN / TH_ACK
TCPS_CLOSING	7	simultaneous close; awaiting ACK	TH_FIN / TH_ACK
TCPS_LAST_ACK	8	received FIN have closed; awaiting ACK	TH_FIN / TH_ACK
TCPS_FIN_WAIT_2	9	have closed; awaiting FIN	TH_ACK
TCPS_TIME_WAIT	10	2MSL wait state after active close	TH_ACK

This figure also shows the tcp\_outflags array, which tells the `tcp_output` function what flags to use when the connection is in a particular state.

Figure 24.16 also shows the numerical values corresponding to each state. These values can be used to check if a connection is in a specific state. For example, the `TCPS_ESTABLISHED` state is defined as:

```
#define TCPS_HAVERCVDSYN (s) (s == 4)
#define TCPS_HAVERCVDFIN (s) (s == 9)
```

Similarly, we'll see that `tcp_notify` handles ICMP messages when the connection is not yet established, that is, when it is in the `TCPS_ESTABLISHED` state.

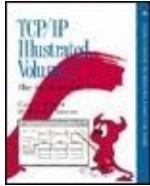
The name `TCPS_HAVERCVDSYN` is correct, but the name `TCPS_HAVERCVDFIN` is misleading. A FIN has been received in the `CLOSE_WAIT`, `CLOSING`, and `LAST_ACK` states. See Chapter 29.

## Half-Close

When a process calls shutdown with a second *close*, TCP sends a FIN but allows the process to socket. (Section 18.5 of Volume 1 contains exa

For example, even though we label the ESTABL the process does a half-close, moving the conn then the FIN\_WAIT\_2 states, data can continue these two states.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.7 TCP Sequence Numbers

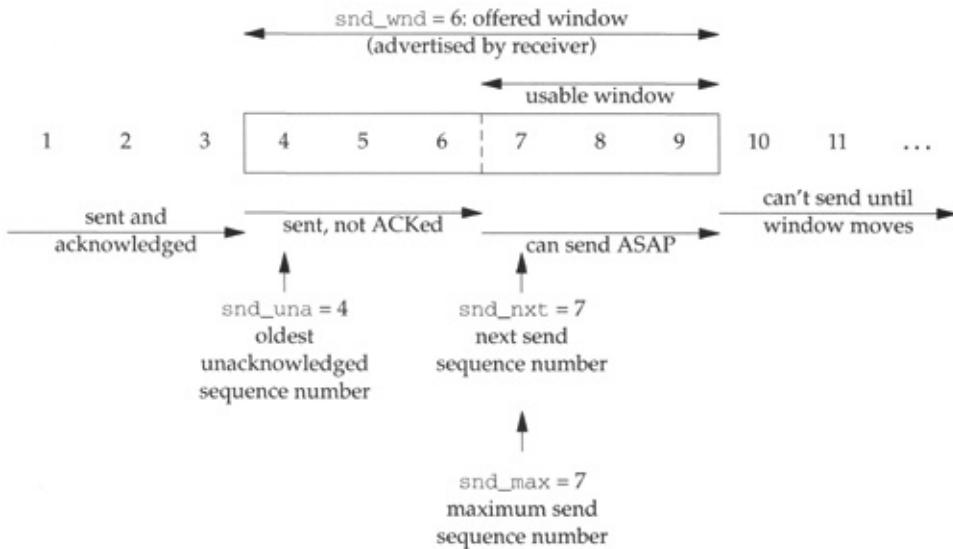
Every byte of data exchanged across a TCP connection, along with the SYN and FIN flags, is assigned a 32-bit *sequence number*. The sequence number field in the TCP header ([Figure 24.10](#)) contains the sequence number of the first byte of data in the segment. The *acknowledgment number* field in the TCP header contains the next sequence number that the sender of the ACK expects to receive, which acknowledges all data bytes through the acknowledgment number minus 1. In other words, the acknowledgment number is the *next* sequence number expected by the sender of the ACK. The

acknowledgment number is valid only if the ACK flag is set in the header. We'll see that TCP always sets the ACK flag except for the first SYN sent by an active open (the SYN\_SENT state; see `tcp_outflags[2]` in [Figure 24.16](#)) and in some RST segments.

Since a TCP connection is *full-duplex*, each end must maintain a set of sequence numbers for both directions of data flow. In the TCP control block ([Figure 24.13](#)) there are 13 sequence numbers: eight for the send direction (the *send sequence space*) and five for the receive direction (the *receive sequence space*).

[Figure 24.17](#) shows the relationship of four of the variables in the send sequence space: `snd_wnd`, `snd_una`, `snd_nxt`, and `snd_max`. In this example we number the bytes 1 through 11.

**Figure 24.17. Example of send sequence space.**



An *acceptable ACK* is one for which the following inequality holds:

$$\text{snd\_una} < \text{acknowledgment field} \leq \text{snd\_max}$$

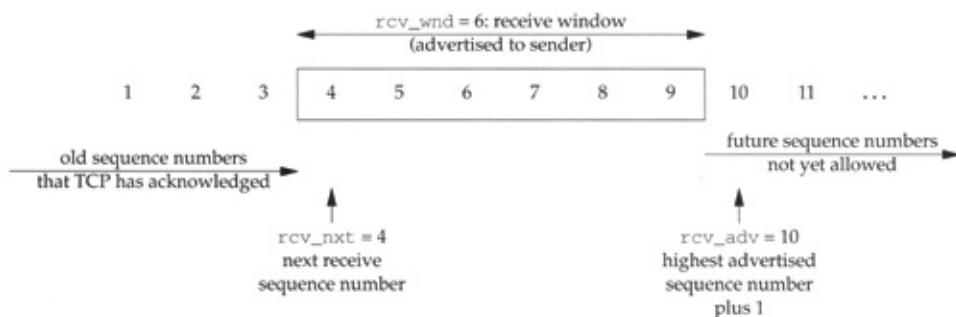
In [Figure 24.17](#) an acceptable ACK has an acknowledgment field of 5, 6, or 7. An acknowledgment field less than or equal to `snd_una` is a duplicate ACK if it acknowledges data that has already been ACKed, or else `snd_una` would not have incremented past those bytes.

We encounter the following test a few times in `tcp_output`, which is true if a segment is being retransmitted:

`snd_nxt < snd_max`

[Figure 24.18](#) shows the other end of the connection in [Figure 24.17](#): the receive sequence space, assuming the segment containing sequence numbers 4, 5, and 6 has not been received yet. We show the three variables `rcv_nxt`, `rcv_wnd`, and `rcv_adv`.

### [Figure 24.18. Example of receive sequence space.](#)



The receiver considers a received segment valid if it contains data within the window, that is, if either of the following two inequalities is true:

`rcv_nxt <= beginning sequence number of segment < rcv_nxt +`

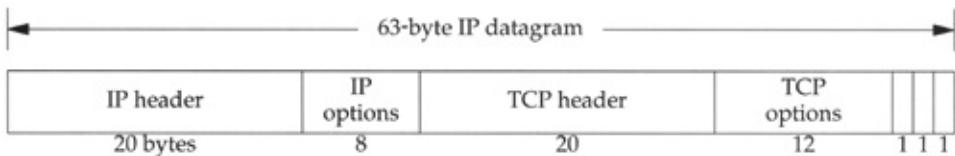
`rcv_wnd`

`rcv_nxt <= ending sequence number of  
segment < rcv_nxt + rcv_wnd`

The beginning sequence number of a segment is just the sequence number field in the TCP header, `ti_seq`. The ending sequence number is the sequence number field plus the number of bytes of TCP data, minus 1.

For example, [Figure 24.19](#) could represent the TCP segment containing the 3 bytes with sequence numbers 4, 5, and 6 in [Figure 24.17](#).

### **Figure 24.19. TCP segment transmitted as an IP datagram.**



We assume that there are 8 bytes of IP options and 12 bytes of TCP options. [Figure 24.20](#) shows the values of the

relevant variables.

## Figure 24.20. Values of variables corresponding to Figure 24.19.

Variable	Value	Description
ip_hl	7	length of IP header + options in 32-bit words (= 28 bytes)
ip_len	63	length of IP datagram in bytes ( $20 + 8 + 20 + 12 + 3$ )
ti_off	8	length of TCP header + options in 32-bit words (= 32 bytes)
ti_seq	4	sequence number of first byte of data
ti_len	3	#bytes of TCP data: $ip\_len - (ip\_hl \times 4) - (ti\_off \times 4)$
	6	sequence number of last byte of data: $ti\_seq + ti\_len - 1$

`ti_len` is not a field that is transmitted in the TCP header. Instead, it is computed as shown in [Figure 24.20](#) and stored in the overlaid IP structure ([Figure 24.12](#)) once the received header fields have been checksummed and verified. The last value in this figure is not stored in the header, but is computed from the other values when needed.

## Modular Arithmetic with Sequence Numbers

A problem that TCP must deal with is that the sequence numbers are from a finite

32-bit number space: 0 through 4,294,967,295. If more than  $2^{32}$  bytes of data are exchanged across a TCP connection, the sequence numbers will be reused. Sequence numbers wrap around from 4,294,967,295 to 0.

Even if less than  $2^{32}$  bytes of data are exchanged, wrap around is still a problem because the sequence numbers for a connection don't necessarily start at 0. The initial sequence number for each direction of data flow across a connection can start anywhere between 0 and 4,294,967,295. This complicates the comparison of sequence numbers. For example, sequence number 1 is "greater than" 4,294,967,295, as we discuss below.

TCP sequence numbers are defined as unsigned longs in `tcp.h`:

```
typedef u_long tcp_seq;
```

The four macros shown in [Figure 24.21](#) compare sequence numbers.

## Figure 24.21. Macros for TCP sequence number comparison.

```
40 #define SEQ_LT(a,b)      ((int)((a)-(b)) < 0)          —tcp_seq.h
41 #define SEQ_LEQ(a,b)     ((int)((a)-(b)) <= 0)
42 #define SEQ_GT(a,b)      ((int)((a)-(b)) > 0)
43 #define SEQ_GEQ(a,b)     ((int)((a)-(b)) >= 0)          —tcp_seq.h
```

## Example Sequence Number Comparisons

Let's look at an example to see how TCP's sequence numbers operate. Assume 3-bit sequence numbers, 0 through 7. [Figure 24.22](#) shows these eight sequence numbers, their 3-bit binary representation, and their two's complement representation. (To form the two's complement take the binary number, convert each 0 to a 1 and vice versa, then add 1.) We show the two's complement because to form  $a - b$  we just add  $a$  to the two's complement of  $b$ .

## Figure 24.22. Example using 3-bit sequence numbers.

x	binary	two's complement	0 - x	1 - x	2 - x
0	000	000	000	001	010
1	001	111	111	000	001
2	010	110	110	111	000
3	011	101	101	110	111
4	100	100	100	101	110
5	101	011	011	100	101
6	110	010	010	011	100
7	111	001	001	010	011

The final three columns of this table are 0 minus x, 1 minus x, and 2 minus x. In these final three columns, if the value is considered to be a *signed* integer (notice the cast to int in all four macros in [Figure 24.21](#)), the value is less than 0 (the SEQ\_LT macro) if the high-order bit is 1, and the value is greater than 0 (the SEQ\_GT macro) if the high-order bit is 0 and the value is not 0. We show horizontal lines in these final three columns to distinguish between the four negative and the four nonnegative values.

If we look at the fourth column of [Figure 24.22](#), (labeled "0 - x"), we see that 0 (i.e., x), is less than 1, 2, 3, and 4 (the high-order bit of the result is 1), and 0 is greater than 5, 6, and 7 (the high-order bit is 0 and the result is not 0). We show

this relationship pictorially in Figure 24.23.

**Figure 24.23. TCP sequence number comparisons for 3-bit sequence numbers.**



Figure 24.24 shows a similar figure using the fifth row of the table (1 - x).

**Figure 24.24. TCP sequence number comparisons for 3-bit sequence numbers.**



Figure 24.25 is another representation of the two previous figures, using circles to reiterate the wrap around of sequence numbers.

**Figure 24.25. Another way to visualize Figures 24.23 and 24.24.**



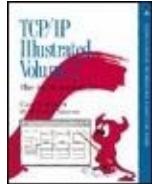
With regard to TCP, these sequence number comparisons determine whether a given sequence number is in the future or in the past (a retransmission). For example, using [Figure 24.24](#), if TCP is expecting sequence number 1 and sequence number 6 arrives, since 6 is less than 1 using the sequence number arithmetic we showed, the data byte is considered a retransmission of a previously received data byte and is discarded. But if sequence number 5 is received, since it is greater than 1 it is considered a future data byte and is saved by TCP, awaiting the arrival of the missing bytes 2, 3, and 4 (assuming byte 5 is within the receive window).

Figure 24.26 is an expansion of the left circle in Figure 24.25, using TCP's 32-bit sequence numbers instead of 3-bit sequence numbers.

### Figure 24.26. Comparisons against 0, using 32-bit sequence numbers.



The right circle in Figure 24.26 is to reiterate that one-half of the 32-bit sequence space uses  $2^{31}$  numbers.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.8 tcp\_init Function

The domaininit function calls TCP's initialization function, `tcp_init` ([Figure 24.27](#)), at system initialization time.

**Figure 24.27. `tcp_init` function.**

```
43 void
44 tcp_init()
. 45 {
46     tcp_iss = 1;           /* wrong */
47     tcb.inp_next = tcb.inp_prev = &tcb;
48     if (max_protohdr < sizeof(struct tcipiphdr))
49         max_protohdr = sizeof(struct tcipiphdr);
50     if (max_linkhdr + sizeof(struct tcipiphdr) > MHLEN)
51         panic("tcp_init");
52 }
```

*tcp\_subr.c*

*tcp\_subr.c*

### Set initial send sequence number (ISS)

46

The initial send sequence number (ISS), `tcp_iss`, is initialized to 1. As the comment indicates, this is wrong. We discuss the implications behind this choice shortly, when we describe TCP's *quiet time*. Compare this to the initialization of the IP identifier in [Figure 7.23](#), which used the time-of-day clock.

## Initialize linked list of TCP Internet PCBs

47

The next and previous pointers in the head PCB (`tcb`) point to itself. This is an empty doubly linked list. The remainder of the `tcb` PCB is initialized to 0 (all uninitialized globals are set to 0), although the only other field used in this head PCB is `inp_lport`, the next TCP ephemeral port number to allocate. The first ephemeral port used by TCP will be 1024, for the reasons described in the solution for [Exercise 22.4](#).

## Calculate maximum protocol header length

48-51

If the maximum protocol header encountered so far is less than 40 bytes, `max_protohdr` is set to 40 (the size of the combined IP and TCP headers, without any options). This variable is described in [Figure 7.17](#). If the sum of `max_linkhdr` (normally 16) and 40 is greater than the amount of data that fits into an mbuf with a packet header (100 bytes, `MHLEN` from [Figure 2.7](#)), the kernel panics ([Exercise 24.2](#)).

## MSL and Quiet Time Concept

TCP requires any host that crashes without retaining any knowledge of the last sequence numbers used on active connections to refrain from sending any TCP segments for one MSL (2 minutes, the quiet time) on reboot. Few TCPs, if any, retain this knowledge over a crash or operator shutdown.

MSL is the *maximum segment lifetime*. Each implementation chooses a value for the MSL. It is the maximum amount of time any segment can exist in the network before being discarded. A connection that is actively closed remains in the CLOSE\_WAIT state ([Figure 24.15](#)) for twice the MSL.

RFC 793 [[Postel 1981c](#)] recommends an MSL of 2 minutes, but Net/3 uses an MSL of 30 seconds (the constant TCPTV\_MSL in [Figure 25.3](#)).

The problem occurs if packets are delayed somewhere in the network (RFC 793 calls these *wandering duplicates*). Assume a Net/3 system starts up, initializes tcp\_iss to 1 (as in [Figure 24.27](#)) and then crashes just after the sequence numbers wrap. We'll see in [Section 25.5](#) that TCP increments tcp\_iss by 128,000 every second, causing the wrap around of the ISS to occur about 9.3 hours after rebooting. Also, tcp\_iss is incremented by 64,000 each time a connect is issued, which can cause the wrap around to occur earlier than 9.3 hours. The following

scenario is one example of how an old segment can incorrectly be delivered to a connection:

**1. A client and server have an established connection. The client's port number is 1024. The client sends a data segment with a starting sequence number of 2. This data segment gets trapped in a routing loop somewhere between the two end points and is not delivered to the server. This data segment becomes a wandering duplicate.**

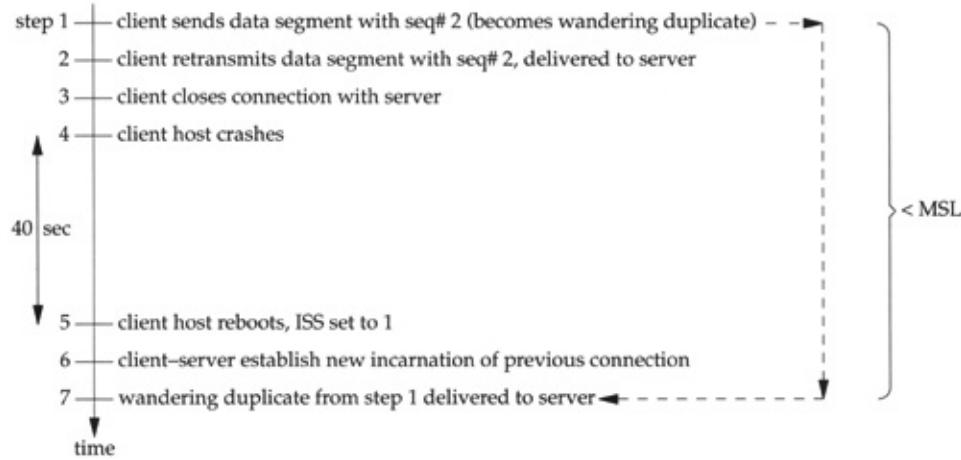
- The client retransmits the data segment starting with sequence number 2, which is delivered to the server.
- The client closes the connection.
- The client host crashes.
- The client host reboots about 40 seconds after crashing, causing TCP to initialize `tcp_iss` to 1 again.
- Another connection is immediately

established by the same client to the same server, using the same socket pair: the client uses 1024 again, and the server uses its well-known port. The client's SYN uses sequence number 1. This new connection using the same socket pair is called a new *incarnation* of the old connection.

- The wandering duplicate from step 1 is delivered to the server, and it thinks this datagram belongs to the new connection, when it is really from the old connection.

[Figure 24.28](#) is a time line of this sequence of steps.

**Figure 24.28. Example of old segment delivered to new incarnation of a connection.**



This problem exists even if the rebooting TCP were to use an algorithm based on its time-of-day clock to choose the ISS on rebooting: regardless of the ISS for the previous incarnation of a connection, because of sequence number wrap it is possible for the ISS after rebooting to nearly equal the sequence number in use before the reboot.

Besides saving the sequence number of all established connections, the only other way around this problem is for the rebooting TCP to be quiet (i.e., not send any TCP segments) for MSL seconds after crashing. Few TCPs do this, however, since it takes most hosts longer than MSL seconds just to reboot.

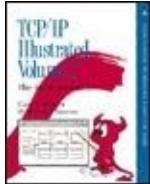
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 24. TCP: Transmission Control Protocol

### 24.9 Summary

This chapter is an introduction to the TCP source code in the six chapters that follow. TCP maintains its own control block for each connection, containing all the variable and state information for the connection.

A state transition diagram is defined for TCP that shows under what conditions TCP moves from one state to another and what segments get sent by TCP for each transition. This diagram shows how connections are established and terminated. We'll refer to this state transition diagram frequently in our description of TCP.

Every byte exchanged across a TCP connection has an associated sequence number, and TCP maintains numerous sequence numbers in the connection control block: some for sending and some for receiving (since TCP is full-duplex). Since these sequence numbers are from a finite 32-bit sequence space, they wrap around from the maximum value back to 0. We explained how the sequence numbers are compared to each other using less-than and greater-than tests, which we'll encounter repeatedly in the TCP code.

Finally, we looked at one of the simplest of the TCP functions, `tcp_init`, which initializes TCP's linked list of Internet PCBs. We also discussed TCP's choice of an initial send sequence number, which is used when actively opening a connection.

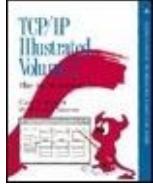
## Exercises

What is the average number of bytes transmitted and received per

**24.1** connection from the statistics in Figure 24.5?

**24.2** Is the kernel panic in tcp\_init reasonable?

Execute netstat -a to see how many  
**24.3** TCP end points your system currently has active.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 25. TCP Timers

[Section 25.1. Introduction](#)

[Section 25.2. Code Introduction](#)

[Section 25.3. `tcp\_canceltimers` Function](#)

[Section 25.4. `tcp\_fasttimo` Function](#)

[Section 25.5. `tcp\_slowtimo` Function](#)

[Section 25.6. `tcp\_timers` Function](#)

[Section 25.7. Retransmission Timer Calculations](#)

[Section 25.8. `tcp\_newtcpcb` Function](#)

[Section 25.9. `tcp\_setpersist` Function](#)

[Section 25.10. `tcp\_xmit\_timer` Function](#)

[Section 25.11. Retransmission Timeout: `tcp\_timers` Function](#)

## Section 25.12. An RTT Example

## Section 25.13. Summary

---

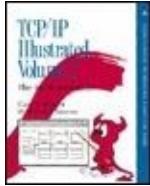
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.1 Introduction

We start our detailed description of the TCP source code by looking at the various TCP timers. We encounter these timers throughout most of the TCP functions.

TCP maintains seven timers for each connection. They are briefly described here, in the approximate order of their occurrence during the lifetime of a connection.

- 1. A *connection-establishment* timer starts when a SYN is sent to establish a new connection. If a response is not received within 75 seconds, the connection**

## **establishment is aborted.**

- A *retransmission* timer is set when TCP sends data. If the data is not acknowledged by the other end when this timer expires, TCP retransmits the data. The value of this timer (i.e., the amount of time TCP waits for an acknowledgment) is calculated dynamically, based on the round-trip time measured by TCP for this connection, and based on the number of times this data segment has been retransmitted. The retransmission timer is bounded by TCP to be between 1 and 64 seconds.
- A *delayed ACK* timer is set when TCP receives data that must be acknowledged, but need not be acknowledged immediately. Instead, TCP waits up to 200 ms before sending the ACK. If, during this 200-ms time period, TCP has data to send on this connection, the pending acknowledgment is sent along with the data (called *piggybacking*).
- A *persist* timer is set when the other end of a connection advertises a window of 0,

stopping TCP from sending data. Since window advertisements from the other end are not sent reliably (that is, ACKs are not acknowledged, only data is acknowledged), there's a chance that a future window update, allowing TCP to send some data, can be lost. Therefore, if TCP has data to send and the other end advertises a window of 0, the persist timer is set and when it expires, 1 byte of data is sent to see if the window has opened. Like the retransmission timer, the persist timer value is calculated dynamically, based on the round-trip time. The value of this is bounded by TCP to be between 5 and 60 seconds.

- A *keepalive* timer can be set by the process using the SO\_KEEPALIVE socket option. If the connection is idle for 2 hours, the keepalive timer expires and a special segment is sent to the other end, forcing it to respond. If the expected response is received, TCP knows that the other host is still up, and TCP won't probe it again until the connection is idle for another 2 hours. Other responses to the keepalive probe tell TCP that the other

host has crashed and rebooted. If no response is received to a fixed number of keepalive probes, TCP assumes that the other end has crashed, although it can't distinguish between the other end being down (i.e., it crashed and has not yet rebooted) and a temporary lack of connectivity to the other end (i.e., an intermediate router or phone line is down).

- A *FIN\_WAIT\_2* timer. When a connection moves from the *FIN\_WAIT\_1* state to the *FIN\_WAIT\_2* state ([Figure 24.15](#)) and the connection cannot receive any more data (implying the process called close, instead of taking advantage of TCP's half-close with shutdown), this timer is set to 10 minutes. When this timer expires it is reset to 75 seconds, and when it expires the second time the connection is dropped. The purpose of this timer is to avoid leaving a connection in the *FIN\_WAIT\_2* state forever, if the other end never sends a FIN. (We don't show this timeout in [Figure 24.15](#).)
- A *TIME\_WAIT* timer, often called the *2MSL* timer. The term *2MSL* means twice

the MSL, the maximum segment lifetime defined in [Section 24.8](#). It is set when a connection enters the TIME\_WAIT state ([Figure 24.15](#)), that is, when the connection is actively closed. Section 18.6 of Volume 1 describes the reasoning for the 2MSL wait state in detail. The timer is set to 1 minute (Net/3 uses an MSL of 30 seconds) when the connection enters the TIME\_WAIT state and when it expires, the TCP control block and Internet PCB are deleted, allowing that socket pair to be reused.

TCP has two timer functions: one is called every 200 ms (the fast timer) and the other every 500 ms (the slow timer). The delayed ACK timer is different from the other six: when the delayed ACK timer is set for a connection it means that a delayed ACK must be sent the next time the 200-ms timer expires (i.e., the elapsed time is between 0 and 200 ms). The other six timers are decremented every 500 ms, and only when the counter reaches 0 does the corresponding action take place.

---

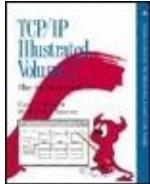
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.2 Code Introduction

The delayed ACK timer is enabled for a connection when the TF\_DELACK flag ([Figure 24.14](#)) is set in the TCP control block. The array `t_timer` in the TCP control block contains four (TCPT\_NTIMERS) counters used to implement the other six timers. The indexes into this array are shown in [Figure 25.1](#). We describe briefly how the six timers (other than the delayed ACK timer) are implemented by these four counters.

**Figure 25.1. Indexes into the `t_timer` array.**

Constant	Value	Description
<code>TCPT_REXMT</code>	0	retransmission timer
<code>TCPT_PERSIST</code>	1	persist timer
<code>TCPT_KEEP</code>	2	keepalive timer <i>or</i> connection-establishment timer
<code>TCPT_2MSL</code>	3	2MSL timer <i>or</i> FIN_WAIT_2 timer

Each entry in the `t_timer` array contains the number of 500-ms clock ticks until the timer expires, with 0 meaning that the timer is not set. Since each timer is a short, if 16 bits hold a short, the maximum timer value is 16,383.5 seconds, or about 4.5 hours.

Notice in [Figure 25.1](#) that four "timer counters" implement six TCP "timers," because some of the timers are mutually exclusive. We'll distinguish between the counters and the timers. The `TCPT_KEEP` counter implements both the keepalive timer and the connection-establishment timer, since the two timers are never used at the same time for a connection.

Similarly, the 2MSL timer and the `FIN_WAIT_2` timer are implemented using the `TCPT_2MSL` counter, since a connection is only in one state at a time.

The first section of [Figure 25.2](#) summarizes the implementation of the seven TCP timers. The second and third

sections of the table show how four of the seven timers are initialized using three global variables from [Figure 24.3](#) and two constants from [Figure 25.3](#). Notice that two of the three globals are used with multiple timers. We've already said that the delayed ACK timer is tied to TCP's 200-ms timer, and we describe how the other two timers are set later in this chapter.

**Figure 25.2. Implementation of the seven TCP timers.**

	conn. estab.	rexmit	delayed ACK	persist	keep- alive	FIN_- WAIT_2	2MSL
t_timer[TCPT_REXMT]		*					
t_timer[TCPT_PERSIST]	*			*			
t_timer[TCPT_KEEP]			*		*		
t_timer[TCPT_2MSL]						*	*
t_flags & TF_DELACK							
tcp_keepidle (2 hr)					*		
tcp_keepintvl (75 sec)					:		
tcp_maxidle (10 min)					:		
2 * TCPTV_MSL (60 sec)	*						
TCPTV_KEEP_INIT (75 sec)							*

**Figure 25.3. Fundamental timer values for the implementation.**

Constant	#500-ms clock ticks	#sec	Description
<code>TCPTV_MSL</code>	60	30	MSL, maximum segment lifetime
<code>TCPTV_MIN</code>	2	1	minimum value of retransmission timer
<code>TCPTV_REXMTMAX</code>	128	64	maximum value of retransmission timer
<code>TCPTV_PERSMIN</code>	10	5	minimum value of persist timer
<code>TCPTV_PERSMAX</code>	120	60	maximum value of persist timer
<code>TCPTV_KEEP_INIT</code>	150	75	connection-establishment timer value
<code>TCPTV_KEEP_IDLE</code>	14400	7200	idle time for connection before first probe (2 hours)
<code>TCPTV_KEEPINTVL</code>	150	75	time between probes when no response
<code>TCPTV_SRTTBASE</code>	0		special value to denote no measurements yet for connection
<code>TCPTV_SRTTDFLT</code>	6	3	default RTT when no measurements yet for connection

[Figure 25.3](#) shows the fundamental timer values for the Net/3 implementation.

[Figure 25.4](#) shows other timer constants that we'll encounter.

## Figure 25.4. Timer constants.

Constant	Value	Description
<code>TCP_LINGERTIME</code>	120	maximum #seconds for <code>SO_LINGER</code> socket option
<code>TCP_MAXRXTSHIFT</code>	12	maximum #retransmissions waiting for an ACK
<code>TCPTV_KEEPCNT</code>	8	maximum #keepalive probes when no response received

The `TCPT_RANGESET` macro, shown in [Figure 25.5](#), sets a timer to a given value, making certain the value is between the specified minimum and maximum.

## Figure 25.5. `TCPT_RANGESET` macro.

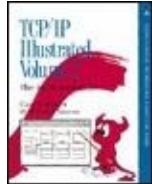
```
102 #define TCPT_RANGESET(tv, value, tvmin, tvmax) { \  
103     (tv) = (value); \  
104     if ((tv) < (tvmin)) \  
105         (tv) = (tvmin); \  
106     else if ((tv) > (tvmax)) \  
107         (tv) = (tvmax); \  
108 }
```

tcp\_timer.h

We see in [Figure 25.3](#) that the retransmission timer and the persist timer have upper and lower bounds, since their values are calculated dynamically, based on the measured round-trip time. The other timers are set to constant values.

There is one additional timer that we allude to in [Figure 25.4](#) but don't discuss in this chapter: the linger timer for a socket, set by the SO\_LINGER socket option. This is a socket-level timer used by the close system call ([Section 15.15](#)). We will see in [Figure 30.12](#) that when a socket is closed, TCP checks whether this socket option is set and whether the linger time is 0. If so, the connection is aborted with an RST instead of TCP's normal close.





TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.3 `tcp_canceltimers` Function

The function `tcp_canceltimers`, shown in [Figure 25.6](#), is called by `tcp_input` when the `TIME_WAIT` state is entered. All four timer counters are set to 0, which turns off the retransmission, persist, keepalive, and `FIN_WAIT_2` timers, before `tcp_input` sets the 2MSL timer.

**Figure 25.6. `tcp_canceltimers` function.**

---

```
107 void
108 tcp_canceltimers(tp)
109 struct tcpcb *tp;
110 {
111     int      i;
112     for (i = 0; i < TCPT_NTIMERS; i++)
113         tp->t_timer[i] = 0;
114 }
```

---

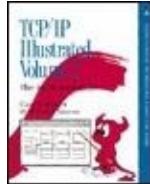
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.4 `tcp_fasttimo` Function

The function `tcp_fasttimo`, shown in [Figure 25.7](#), is called by `pr_fasttimo` every 200 ms. It handles only the delayed ACK timer.

**Figure 25.7. `tcp_fasttimo` function, which is called every 200 ms.**

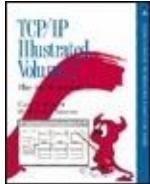
```
41 void
42 tcp_fasttimo()
43 {
44     struct inpcb *inp;
45     struct tcpcb *tp;
46     int     s = splnet();
47
48     inp = tcb.inp_next;
49     if (inp)
50         for (; inp != &tcb; inp = inp->inp_next)
51             if ((tp = (struct tcpcb *) inp->inp_ppcb) &&
52                 (tp->t_flags & TF_DELACK)) {
53                 tp->t_flags &= ~TF_DELACK;
54                 tp->t_flags |= TF_ACKNOW;
55                 tcpstat.tcps_delack++;
56                 (void) tcp_output(tp);
57             }
58     splx(s);
```

*tcp\_timer.c*

Each Internet PCB on the TCP list that has a corresponding TCP control block is checked. If the TF\_DELACK flag is set, it is cleared and the TF\_ACKNOW flag is set instead. `tcp_output` is called, and since the TF\_ACKNOW flag is set, an ACK is sent.

How can TCP have an Internet PCB on its PCB list that doesn't have a TCP control block (the test at line 50)? When a socket is created (the PRU\_ATTACH request, in response to the socket system call) we'll see in [Figure 30.11](#) that the creation of the Internet PCB is done first, followed by the creation of the TCP control block. Between these two operations a high-priority clock interrupt can occur ([Figure 1.13](#)), which calls `tcp_fasttimo`.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.5 `tcp_slowtimo` Function

The function `tcp_slowtimo`, shown in [Figure 25.8](#), is called by `pr_slowtimo` every 500 ms. It handles the other six TCP timers: connection establishment, retransmission, persist, keepalive, `FIN_WAIT_2`, and `2MSL`.

**Figure 25.8. `tcp_slowtimo` function,  
which is called every 500 ms.**

---

```

64 void
65 tcp_slowtimo()
66 {
67     struct inpcb *ip, *ipnxt;
68     struct tcpcb *tp;
69     int      s = splnet();
70     int      i;

71     tcp_maxidle = TCPTV_KEEPCNT * tcp_keepintvl;
72     /*
73      * Search through tcb's and update active timers.
74      */
75     ip = tcb.inp_next;
76     if (ip == 0) {
77         splx(s);
78         return;
79     }
80     for (; ip != &tcb; ip = ipnxt) {
81         ipnxt = ip->inp_next;
82         tp = intotcpcb(ip);
83         if (tp == 0)
84             continue;
85         for (i = 0; i < TCPT_NTIMERS; i++) {
86             if (tp->t_timer[i] && --tp->t_timer[i] == 0) {
87                 (void) tcp_usrreq(tp->t_inpcb->inp_socket,
88                                     PRU_SLOWTIMO, (struct mbuf *) 0,
89                                     (struct mbuf *) i, (struct mbuf *) 0);
90                 if (ipnxt->inp_prev != ip)
91                     goto tpgone;
92             }
93         }
94         tp->t_idle++;
95         if (tp->t_rtt)
96             tp->t_rtt++;
97     tpgone:
98     ;
99     }
100    tcp_iss += TCP_ISSINCR / PR_SLOWHZ;      /* increment iss */
101    tcp_now++;                            /* for timestamps */
102    splx(s);
103 }

```

---

71

tcp\_maxidle is initialized to 10 minutes. This is the maximum amount of time TCP will send keepalive probes to another host, waiting for a response from that host. This variable is also used with the FIN\_WAIT\_2 timer, as we describe in [Section 25.6](#). This initialization statement could be moved to tcp\_init, since it only needs to be

evaluated when the system is initialized (see [Exercise 25.2](#)).

## Check each timer counter in all TCP control blocks

72-89

Each Internet PCB on the TCP list that has a corresponding TCP control block is checked. Each of the four timer counters for each connection is tested, and if nonzero, the counter is decremented. When the timer reaches 0, a PRU\_SLOWTIMO request is issued. We'll see that this request calls the function `tcp_timers`, which we describe later in this chapter.

The fourth argument to `tcp_usrreq` is a pointer to an mbuf. But this argument is actually used for different purposes when the mbuf pointer is not required. Here we see the index `i` is passed, telling the request which timer has expired. The funny-looking cast of `i` to an mbuf pointer is to avoid a compile-time error.

## **Check if TCP control block has been deleted**

**90-93**

Before examining the timers for a control block, a pointer to the next Internet PCB is saved in ipnxt. Each time the PRU\_SLOWTIMO request returns, tcp\_slowtimo checks whether the next PCB in the TCP list still points to the PCB that's being processed. If not, it means the control block has been deleted perhaps the 2MSL timer expired or the retransmission timer expired and TCP is giving up on this connection causing a jump to tpgone, skipping the remaining timers for this control block, and moving on to the next PCB.

## **Count idle time**

**94**

t\_idle is incremented for the control block. This counts the number of 500-ms clock ticks since the last segment was received

on this connection. It is set to 0 by `tcp_input` when a segment is received on the connection and used for three purposes: (1) by the keepalive algorithm to send a probe after the connection is idle for 2 hours, (2) to drop a connection in the `FIN_WAIT_2` state that is idle for 10 minutes and 75 seconds, and (3) by `tcp_output` to return to the slow start algorithm after the connection has been idle for a while.

## Increment RTT counter

95-96

If this connection is timing an outstanding segment, `t_rtt` is nonzero and counts the number of 500-ms clock ticks until that segment is acknowledged. It is initialized to 1 by `tcp_output` when a segment is transmitted whose RTT should be timed. `tcp_slowtimo` increments this counter.

## Increment initial send sequence number

100

`tcp_iss` was initialized to 1 by `tcp_init`. Every 500 ms it is incremented by 64,000: 128,000 (`TCP_ISSINCR`) divided by 2 (`PR_SLOWHZ`). This is a rate of about once every 8 microseconds, although `tcp_iss` is incremented only twice a second. We'll see that `tcp_iss` is also incremented by 64,000 each time a connection is established, either actively or passively.

RFC 793 specifies that the initial sequence number should increment roughly every 4 microseconds, or 250,000 times a second. The Net/3 value increments at about one-half this rate.

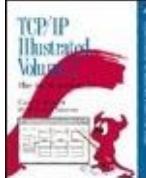
## Increment RFC 1323 timestamp value

101

`tcp_now` is initialized to 0 on bootstrap and incremented every 500 ms. It is used by the timestamp option defined in RFC 1323 [[Jacobson, Braden, and Borman 1992](#)], which we describe in Section 26.6.

Notice that if there are no TCP connections active on the host (tcb.inp\_next is null), neither tcp\_iss nor tcp\_now is incremented. This would occur only when the system is being initialized, since it would be rare to find a Unix system attached to a network without a few TCP servers active.

---



TCP/IP Illustrated, Volume 2: The Implementation  
By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.6 tcp\_timers Function

The function `tcp_timers` is called by TCP's `PRU_SLOWTIMO` request (Figure 30.10):

```
case PRU_SLOWTIMO:  
    tp = tcp_timers(tp, (int)nam)
```

when any one of the four TCP timer counters reaches 0 (Figure 25.8).

The structure of the function is a switch statement with one case per timer, as outlined in Figure 25.9.

**Figure 25.9. `tcp_timers` function: general organization.**

```

120 struct tcpcb *
121 tcp_timers(tp, timer)
122 struct tcpcb *tp;
123 int timer;
124 {
125     int rexmt;
126     switch (timer) {
127         /* switch cases */
128     }
129     return (tp);
130 }

```

tcp\_timer.c

We now discuss three of the four timer counter (five of TCP's timers), saving the retransmission timer for [Section 25.11](#).

## FIN\_WAIT\_2 and 2MSL Timers

TCP's TCPT\_2MSL counter implements two of TCP's timers.

- 1. FIN\_WAIT\_2 timer. When `tcp_input` moves from the FIN\_WAIT\_1 state to the FIN\_WAIT\_2 state and the socket cannot receive any more data (implying the process called `close`, instead of taking advantage of TCP's half-close with `shutdown`), the FIN\_WAIT\_2 timer is set to 10 minutes (`tcp_maxidle`). We see that this prevents the connection from staying in the FIN\_WAIT\_2 state forever.**

- 2MSL timer. When TCP enters the TIME\_WAIT state, the 2MSL timer is set to 60 seconds (TCPPTV\_MSL times 2).

[Figure 25.10](#) shows the case for the 2MSL timer executed when the timer reaches 0.

## Figure 25.10. `tcp_timers` function: expiration of 2MSL timer counter.

```

127      /*
128      * 2 MSL timeout in shutdown went off.  If we're closed but
129      * still waiting for peer to close and connection has been idle
130      * too long, or if 2MSL time is up from TIME_WAIT, delete connection
131      * control block.  Otherwise, check again in a bit.
132      */
133  case TCPT_2MSL:
134      if (tp->t_state != TCPS_TIME_WAIT &&
135          tp->t_idle <= tcp_maxidle)
136          tp->t_timer[TCPT_2MSL] = tcp_keepintvl;
137      else
138          tp = tcp_close(tp);
139      break;

```

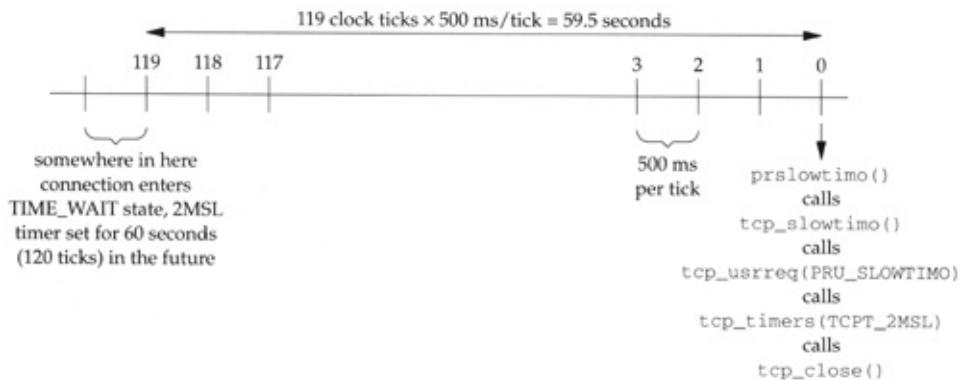
## 2MSL timer

127-139

The puzzling logic in the conditional is because the two different uses of the TCPT\_2MSL counter are intermixed ([Exercise 25.4](#)). Let's first look at the TIME\_WAIT state. When the timer expires after 60 seconds, `tcp_close` is called and the

control blocks are released. We have the scenario shown in [Figure 25.11](#). This figure shows the series of function calls that occurs when the 2MSL timer expires. We also see that setting one of the timers for  $N$  seconds in the future ( $2 \times N$  ticks), causes the timer to expire somewhere between  $2 \times N - 1$  and  $2 \times N$  ticks in the future, since the time until the first decrement of the counter is between 0 and 500 ms in the future.

**Figure 25.11. Setting and expiration of 2MSL timer in TIME\_WAIT state.**



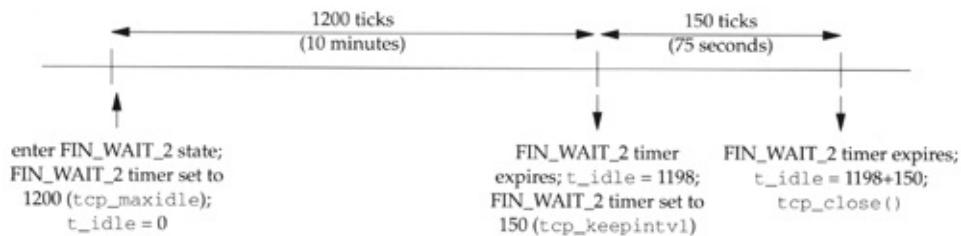
## FIN\_WAIT\_2 timer

127-139

If the connection state is not TIME\_WAIT, the

TCPT\_2MSL counter is the FIN\_WAIT\_2 timer. As soon as the connection has been idle for more than 10 minutes (tcp\_maxidle) the connection closed. But if the connection has been idle for less than or equal to 10 minutes, the FIN\_WAIT\_2 timer is reset for 75 seconds in the future. [Figure 25.12](#) shows the typical scenario

**Figure 25.12. FIN\_WAIT\_2 timer to avoid infinite wait in FIN\_WAIT\_2 state.**



The connection moves from the FIN\_WAIT\_1 state to the FIN\_WAIT\_2 state on the receipt of an ACK ([Figure 24.15](#)). Receiving this ACK sets  $t_{idle}$  to 0 and the FIN\_WAIT\_2 timer is set to 1200 (tcp\_maxidle). In [Figure 25.12](#) we show the up arrow just to the right of the tick mark starting the 10-minute period, to reiterate that the first decrement of the counter occurs between 0 and 500 ms after the counter is set. After 1199 ticks the timer expires, but since  $t_{idle}$  is incremented *after* the test and

decrement of the four counters in [Figure 25.8](#), `t_idle` is 1198. (We assume the connection is idle for this 10-minute period.) The comparison of 1198 as less than or equal to 1200 is true, so the `FIN_WAIT_2` timer is set to 150 (`tcp_keepintvl`). When the timer expires again in 75 seconds, assuming the connection is still idle, `t_idle` is now 1348, the test is false, and `tcp_close` is called.

The reason for the 75-second timeout after the first 10-minute timeout is as follows: a connection in the `FIN_WAIT_2` state is not dropped until the connection has been idle for *more than* 10 minutes. There's no reason to test `t_idle` until at least 10 minutes have expired, but once this time has passed, the value of `t_idle` is checked every 75 seconds. Since a duplicate segment could be received, say a duplicate of the ACK that moved the connection from the `FIN_WAIT_1` state to the `FIN_WAIT_2` state, the 10-minute wait is restarted when the segment is received (since `t_idle` will be set to 0).

Terminating an idle connection after more than 10 minutes in the `FIN_WAIT_2` state violates the protocol specification, but this is practical. In the `FIN_WAIT_2` state the

process has called close, all outstanding data on the connection has been sent and acknowledged, the other end has acknowledged the FIN, and TCP is waiting for the process at the other end of the connection to issue its close. If the other process never closes its end of the connection, our end can remain in the FIN\_WAIT\_2 forever. A counter should be maintained for the number of connections terminated for this reason, to see how often this occurs.

## Persist Timer

Figure 25.13 shows the case for when the persist timer expires.

### Figure 25.13. `tcp_timers` function: expiration of persist timer.

```
210      /*          _____tcp_timer.c
211      * Persistence timer into zero window.
212      * Force a byte to be output, if possible.
213      */
214  case TCPT_PERSIST:
215      tcpstat.tcpst_persisttimeo++;
216      tcp_setpersist(tp);
217      tp->t_force = 1;
218      (void) tcp_output(tp);
219      tp->t_force = 0;
220      break;          _____tcp_timer.c
```

## Force window probe segment

210-220

When the persist timer expires, there is data to send on the connection but TCP has been stopped by the other end's advertisement of a zero-sized window. `tcp_setpersist` calculates the next value for the persist timer and stores it in the `TCPT_PERSIST` counter. The flag `t_force` is set to 1, forcing `tcp_output` to send 1 byte, even though the window advertised by the other end is 0.

**Figure 25.14** shows typical values of the persist timer for a LAN, assuming the retransmission timeout for the connection is 1.5 seconds (see Figure 22.1 of Volume 1).

**Figure 25.14. Time line of persist timer when probing a zero window.**



Once the value of the persist timer reaches 60 seconds, TCP continues sending window probes

every 60 seconds. The reason the first two values are both 5, and not 1.5 and 3, is that the persist timer is lower bounded at 5 seconds. It is also upper bounded at 60 seconds. The multiplication of each value by 2 to give the next value is called an *exponential backoff*, and we describe how it is calculated in Section 25.9.

## Connection Establishment and Keepalive Timers

TCP's TCPTV\_KEEP counter implements two timers:

- 1. When a SYN is sent, the connection-establishment timer is set to 75 seconds (TCPTV\_KEEP\_INIT). This happens when connect is called, putting a connection into the SYN\_SENT state (active open), or when a connection moves from the LISTEN to the SYN\_RCVD state (passive open). If the connection doesn't enter the ESTABLISHED state within 75 seconds, the connection is dropped.**
- When a segment is received on a connection, tcp\_input resets the keepalive timer for that

connection to 2 hours (`tcp_keepidle`), and the `t_idle` counter for the connection is reset to 0. This happens for every TCP connection on the system, whether the keepalive option is enable for the socket or not. If the keepalive timer expires (2 hours after the last segment was received on the connection), and if the socket option is set, a keepalive probe is sent to the other end. If the timer expires and the socket option is not set, the keepalive timer is just reset for 2 hours in the future.

[Figure 25.16](#) shows the case for TCP's `TCPT_KEEP` counter.

## Connection-establishment timer expires after 75 seconds

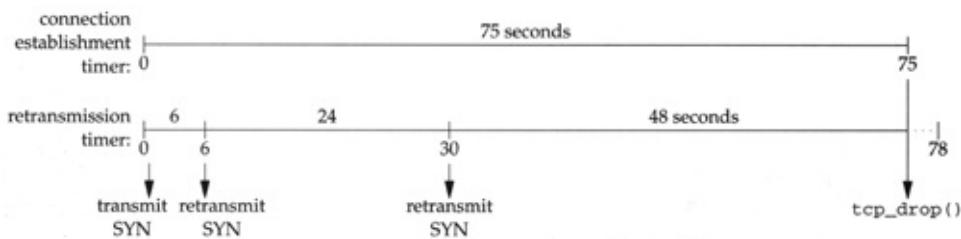
221-228

If the state is less than `ESTABLISHED` ([Figure 24.16](#)), the `TCPT_KEEP` counter is the connection-establishment timer. At the label `dropit`, `tcp_drop` is called to terminate the connection attempt with an error of `ETIMEDOUT`. We'll see that this error is the default error if, for example, a soft error such as an ICMP host

unreachable was received on the connection, the error returned to the process will be changed to EHOSTUNREACH instead of the default.

In [Figure 30.4](#) we'll see that when TCP sends a SYN, two timers are initialized: the connection-establishment timer as we just described, with a value of 75 seconds, and the retransmission timer, to cause the SYN to be retransmitted if no response is received. [Figure 25.15](#) shows these two timers.

## Figure 25.15. Connection-establishment timer and retransmission timer after SYN is sent.



The retransmission timer is initialized to 6 seconds for a new connection ([Figure 25.19](#)), and successive values are calculated to be 24 and 48 seconds. We describe how these values are calculated in [Section 25.7](#). The retransmission timer causes the SYN to be transmitted a total of three times, at times 0, 6, and 30.

and 30. At time 75, 3 seconds before the retransmission timer would expire again, the connection-establishment timer expires, and `tcp_drop` terminates the connection attempt.

**Figure 25.16. `tcp_timers` function: expiration of keepalive timer.**

```
221      /*
222       * Keep-alive timer went off; send something
223       * or drop connection if idle for too long.
224      */
225  case TCPT_KEEP:
226      tcpstat.tcps_keeptimeo++;
227      if (tp->t_state < TCPS_ESTABLISHED)
228          goto dropit;           /* connection establishment timer */

229      if (tp->t_inpcb->inp_socket->so_options & SO_KEEPALIVE &&
230          tp->t_state <= TCPS_CLOSE_WAIT) {
231          if (tp->t_idle >= tcp_keepidle + tcp_maxidle)
232              goto dropit;
233          /*
234           * Send a packet designed to force a response
235           * if the peer is up and reachable:
236           * either an ACK if the connection is still alive,
237           * or an RST if the peer has closed the connection
238           * due to timeout or reboot.
239           * Using sequence number tp->snd_una-1
240           * causes the transmitted zero-length segment
241           * to lie outside the receive window;
242           * by the protocol spec, this requires the
243           * correspondent TCP to respond.
244           */
245          tcpstat.tcps_keepprobe++;
246          tcp_respond(tp, tp->t_template, (struct mbuf *) NULL,
247                      tp->rcv_nxt, tp->snd_una - 1, 0);
248          tp->t_timer[TCPT_KEEP] = tcp_keepintvl;
249      } else
250          tp->t_timer[TCPT_KEEP] = tcp_keepidle;
251      break;
252  dropit:
253      tcpstat.tcps_kepdrops++;
254      tp = tcp_drop(tp, ETIMEDOUT);
255      break;
```

**Keepalive timer expires after 2 hours of idle time**

229-230

This timer expires after 2 hours of idle time on every connection, not just ones with the SO\_KEEPALIVE socket option enabled. If the socket option is set, probes are sent only if the connection is in the ESTABLISHED or CLOSE\_WAIT states ([Figure 24.15](#)). Once the process calls close (the states greater than CLOSE\_WAIT), keepalive probes are not sent, even if the connection is idle for 2 hours.

## Drop connection when no response

231-232

If the total idle time for the connection is greater than or equal to 2 hours (tcp\_keepidle) plus 10 minutes (tcp\_maxidle), the connection dropped. This means that TCP has sent its limit of nine keepalive probes, 75 seconds apart (tcp\_keepintvl), with no response. One reason TCP must send multiple keepalive probes before considering the connection dead is that the ACKs sent in response do not contain data and therefore are not reliably transmitted by TCP. A ACK that is a response to a keepalive probe car

get lost.

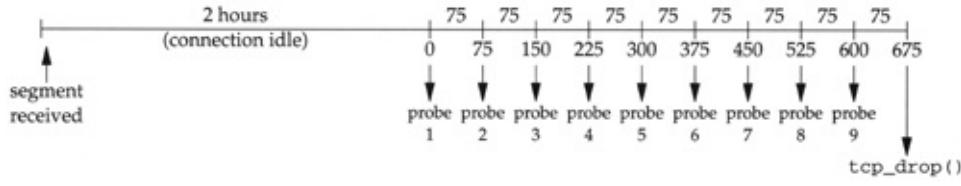
## Send a keepalive probe

233-248

If TCP hasn't reached the keepalive limit, `tcp_respond` sends a keepalive packet. The acknowledgment field of the keepalive packet (the fourth argument to `tcp_respond`) contains `rcv_nxt`, the next sequence number expected on the connection. The sequence number field of the keepalive packet (the fifth argument) deliberately contains `snd_una` minus 1, which is the sequence number of a byte of data that the other end has already acknowledged ([Figure 24.17](#)). Since this sequence number is outside the window, the other end must respond with an ACK, specifying the next sequence number it expects.

[Figure 25.17](#) summarizes this use of the keepalive timer.

**Figure 25.17. Summary of keepalive timer to detect unreachability of other end.**



The nine keepalive probes are sent every 75 seconds, starting at time 0, through time 600. At time 675 (11.25 minutes after the 2-hour timer expired) the connection is dropped. Notice that nine keepalive probes are sent, even though the constant TCPTV\_KEEP\_CNT ([Figure 25.4](#)) is 8. This is because the variable `t_idle` is incremented in [Figure 25.8](#) after the timer is decremented, compared to 0, and possibly handled. When `tcp_input` receives a segment on a connection, it sets the keepalive timer to 14400 (`tcp_keepidle`) and `t_idle` to 0. The next time `tcp_slowtimo` is called, the keepalive timer is decremented to 14399 and `t_idle` is incremented to 1. About 2 hours later, when the keepalive timer is decremented from 1 to 0 and `tcp_timers` is called, the value of `t_idle` will be 14399. We can build the table in [Figure 25.18](#) to see the value of `t_idle` each time `tcp_timers` is called.

**Figure 25.18. The value of `t_idle` when `tcp_timers` is called for keepalive processing**

probe#	time in Figure 25.17	t_idle
1	0	14399
2	75	14549
3	150	14699
4	225	14849
5	300	14999
6	375	15149
7	450	15299
8	525	15449
9	600	15599
	675	15749

The code in [Figure 25.16](#) is waiting for t\_idle to be greater than or equal to 15600 (`tcp_keepidle + tcp_maxidle`) and that only happens at time 675 in [Figure 25.17](#), after nine keepalive probe have been sent.

## Reset keepalive timer

249-250

If the socket option is not set or the connection state is greater than `CLOSE_WAIT`, the keepalive timer for this connection is reset to 2 hours (`tcp_keepidle`).

Unfortunately the counter `tcps_keepdrops` (line 253) counts both uses of the `TCPT_KEEF` counter: the connection-establishment timer and the keepalive timer.

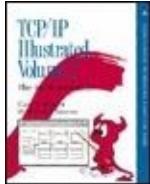
---

**Team-Fly**



[Previous](#)

T



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.7 Retransmission Timer Calculations

The timers that we've described so far in this chapter have fixed times associated with them: 200 ms for the delayed ACK timer, 75 seconds for the connection-establishment timer, 2 hours for the keepalive timer, and so on. The final two timers that we describe, the retransmission timer and the persist timer, have values that depend on the measured RTT for the connection. Before going through the source code that calculates and sets these timers we need to understand how TCP measures the RTT for a connection.

Fundamental to the operation of TCP is setting a retransmission timer when a segment is transmitted and an ACK is required from the other end. If the ACK is not received when the retransmission timer expires, the segment is retransmitted. TCP requires an ACK for data segments but does not require an ACK for a segment without data (i.e., a pure ACK segment). If the calculated retransmission timeout is too small, it can expire prematurely, causing needless retransmissions. If the calculated value is too large, after a segment is lost, additional time is lost before the segment is retransmitted, degrading performance. Complicating this is that the round-trip times between two hosts can vary widely and dynamically over the course of a connection.

TCP in Net/3 calculates the retransmission timeout (*RTO*) by measuring the round-trip time (*nticks*) of data segments and keeping track of the smoothed RTT estimator (*srtt*) and a smoothed mean deviation estimator (*rttvar*). The mean deviation is a good approximation of the

standard deviation, but easier to compute since, unlike the standard deviation, the mean deviation does not require square root calculations. [Jacobson 1988b] provides additional details on these RTT measurements, which lead to the following equations:

```
delta = nticks - srtt  
srtt ← srtt + g × delta  
rttvar ← rttvar + h(|delta| - rttvar)  
RTO = srtt + 4 × rttvar
```

*delta* is the difference between the measured round trip just obtained (*nticks*) and the current smoothed RTT estimator (*srtt*). *g* is the gain applied to the RTT estimator and equals 1/8. *h* is the gain applied to the mean deviation estimator and equals 1/4. The two gains and the multiplier 4 in the RTO calculation are purposely powers of 2, so they can be calculated using shift operations instead of multiplying or dividing.

[Jacobson 1988b] specified  $2 \times rttvar$  in the calculation of *RTO*, but after further

research, [Jacobson 1990d] changed the value to  $4 \times rttvar$ , which is what appeared in the Net/1 implementation.

We now describe the variables and calculations used to calculate TCP's retransmission timer, as we'll encounter them throughout the TCP code. [Figure 25.19](#) lists the variables in the control block related to the retransmission timer.

### **Figure 25.19. Control block variables for calculation of retransmission timer.**

tcpcb member	Units	tcp_newtcpcb initial value	#sec	Description
t_srtt	ticks $\times 8$	0		smoothed RTT estimator: $srtt \times 8$
t_rttvar	ticks $\times 4$	24	3	smoothed mean deviation estimator: $rttvar \times 4$
t_rxtcur	ticks	12	6	current retransmission timeout: RTO
t_rttmin	ticks	2	1	minimum value for retransmission timeout
t_rxtshift	n.a.	0		index into <code>tcp_backoff[]</code> array (exponential backoff)

We show the `tcp_backoff` array at the end of [Section 25.9](#). The `tcp_newtcpcb` function sets the initial values for these variables, and we cover it in the next section. The term *shift* in the variable `t_rxtshift` and its limit `TCP_MAXRXTSHIFT` is not entirely accurate. The former is not used for bit shifting, but as [Figure 25.19](#)

indicates, it is an index into an array.

The confusing part of TCP's timeout calculations is that the two smoothed estimators maintained in the C code (`t_srtt` and `t_rttvar`) are fixed-point integers, instead of floating-point values. This is done to avoid floating-point calculations within the kernel, but it complicates the code.

To keep the scaled and unsealed variables distinct, we'll use the italic variables *srtt* and *rttvar* to refer to the unsealed variables in the earlier equations, and `t_srtt` and `t_rttvar` to refer to the scaled variables in the TCP control block.

Figure 25.20 shows four constants we encounter, which define the scale factors of 8 for `t_srtt` and 4 for `t_rttvar`.

**Figure 25.20. Multipliers and shifts for RTT estimators.**

Constant	Value	Description
<i>TCP_RTT_SCALE</i>	8	multiplier: $t_{\text{sr}tt} = sr{tt} \times 8$
<i>TCP_RTT_SHIFT</i>	3	shift: $t_{\text{sr}tt} = sr{tt} \ll 3$
<i>TCP_RTTVAR_SCALE</i>	4	multiplier: $t_{\text{rtt}var} = rt{t}var \times 4$
<i>TCP_RTTVAR_SHIFT</i>	2	shift: $t_{\text{rtt}var} = rt{t}var \ll 2$

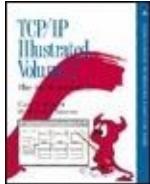
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.8 `tcp_newtcpcb` Function

A new TCP control block is allocated and initialized by `tcp_newtcpcb`, shown in [Figure 25.21](#). This function is called by TCP's PRU\_ATTACH request when a new socket is created ([Figure 30.2](#)). The caller has previously allocated an Internet PCB for this connection, pointed to by the argument `inp`. We present this function now because it initializes the TCP timer variables.

**Figure 25.21. `tcp_newtcpcb` function:  
create and initialize a new TCP control  
block.**

---

```

167 struct tcppcb *
168 tcp_newtcpcb(inp)
169 struct inpcb *inp;
170 {
171     struct tcppcb *tp;
172     tp = malloc(sizeof(*tp), M_PCB, M_NOWAIT);
173     if (tp == NULL)
174         return ((struct tcppcb *) 0);
175     bzero((char *) tp, sizeof(struct tcppcb));
176     tp->seg_next = tp->seg_prev = (struct tcphdr *) tp;
177     tp->t_maxseg = tcp_mssdflt;
178     tp->t_flags = tcp_do_rfc1323 ? (TF_REQ_SCALE | TF_REQ_TSTMP) : 0;
179     tp->t_inpcb = inp;
180     /*
181      * Init srtt to TCPTV_SRTTBASE (0), so we can tell that we have no
182      * rtt estimate. Set rttvar so that srtt + 2 * rttvar gives
183      * reasonable initial retransmit time.
184     */
185     tp->t_srtt = TCPTV_SRTTBASE;
186     tp->t_rttvar = tcp_rttdflt * PR_SLOWHZ << 2;
187     tp->t_rttmin = TCPTV_MIN;
188     TCPT_RANGESET(tp->t_rxcur,
189                   ((TCPTV_SRTTBASE >> 2) + (TCPTV_SRTTDFLT << 2)) >> 1,
190                   TCPTV_MIN, TCPTV_REXMTMAX);
191     tp->snd_cwnd = TCP_MAXWIN << TCP_MAX_WINSHIFT;
192     tp->snd_ssthresh = TCP_MAXWIN << TCP_MAX_WINSHIFT;
193     inp->inp_ip.ip_ttl = ip_defttl;
194     inp->inp_ppcb = (caddr_t) tp;
195     return (tp);
196 }

```

---

tcp\_subr.c

## 167-175

The kernel's malloc function allocates memory for the control block, and bzero sets it to 0.

## 176

The two variables seg\_next and seg\_prev point to the reassembly queue for out-of-order segments received for this connection. We discuss this queue in detail in [Section 27.9](#).

177-179

The maximum segment size to send, `t_maxseg`, defaults to 512 (`tcp_mssdflt`). This value can be changed by the `tcp_mss` function after an MSS option is received from the other end. (TCP also sends an MSS option to the other end when a new connection is established.) The two flags `TF_REQ_SCALE` and `TF_REQ_TSTMP` are set if the system is configured to request window scaling and timestamps as defined in RFC 1323 (the global `tcp_do_rfc1323` from [Figure 24.3](#), which defaults to 1). The `t_inpcb` pointer in the TCP control block is set to point to the Internet PCB passed in by the caller.

180-185

The four variables `t_srtt`, `t_rttvar`, `t_rttmin`, and `t_rxcur`, described in [Figure 25.19](#), are initialized. First, the smoothed RTT estimator `t_srtt` is set to 0 (`TCPTV_SRTTBASE`), which is a special value that means no RTT measurements have been made yet for this connection. `tcp_xmit_timer` recognizes this special

value when the first RTT measurement is made.

186-187

The smoothed mean deviation estimator `t_rttvar` is set to 24: 3 (`tcp_rttdflt`, from [Figure 24.3](#)) times 2 (`PR_SLOWHZ`) multiplied by 4 (the left shift of 2 bits). Since this scaled estimator is 4 times the variable `rttvar`, this value equals 6 clock ticks, or 3 seconds. The minimum *RTO*, stored in `t_rttmin`, is 2 ticks (`TCPTV_MIN`).

188-190

The current *RTO* in clock ticks is calculated and stored in `t_rxtdcur`. It is bounded by a minimum value of 2 ticks (`TCPTV_MIN`) and a maximum value of 128 ticks (`TCPTV_REXMTMAX`). The value calculated as the second argument to `TCPT_RANGESET` is 12 ticks, or 6 seconds. This is the first *RTO* for the connection.

Understanding these C expressions involving the scaled RTT estimators can be a challenge. It helps to start with the

unsealed equation and substitute the scaled variables. The unsealed equation we're solving is

$$RTO = srtt + 2 \times rttvar$$

where we use the multiplier of 2 instead of 4 to calculate the first *RTO*.

The use of the multiplier 2 instead of 4 appears to be a leftover from the original 4.3BSD Tahoe code [[Paxson 1994](#)].

Substituting the two scaling relationships

$$t_{srtt} = 8 \times srtt$$

$$t_{rttvar} = 4 \times rttvar$$

we get

$$\begin{aligned} RTO &= \frac{t_{srtt}}{8} + 2 \times \frac{t_{rttvar}}{4} \\ &= \frac{\frac{t_{srtt}}{4} + t_{rttvar}}{2} \end{aligned}$$

which is the C code for the second argument to TCPT\_RANGESET. In this code the variable `t_rttvar` is not used; the constant `TCPTV_SRTTDFLT`, whose value is 6 ticks, is used instead, and it must be multiplied by 4 to have the same scale as `t_rttvar`.

191-192

The congestion window (`snd_cwnd`) and slow start threshold (`snd_ssthresh`) are set to 1,073,725,440 (approximately one gigabyte), which is the largest possible TCP window if the window scale option is in effect. (Slow start and congestion avoidance are described in Section 21.6 of Volume 1.) It is calculated as the maximum value for the window size field in the TCP header (65535, `TCP_MAXWIN`) times  $2^{14}$ , where 14 is the maximum value for the window scale factor (`TCP_MAX_WINSHIFT`). We'll see that when a SYN is sent or received on the connection, `tcp_mss` resets `snd_cwnd` to a single segment.

193-194

The default IP TTL in the Internet PCB is set to 64 (ip\_defttl) and the PCB is set to point to the new TCP control block.

Not shown in this code is that numerous variables, such as the shift variable t\_rxtshift, are implicitly initialized to 0 since the control block is initialized by bzero.

---

## Chapter 25. TCP Timers

---

### 25.9 tcp\_setpersist Function

The next function we look at that uses TCP's retransmission calculations is `tcp_setpersist`. In [Figure 25.13](#) we see what happens when the persist timer expired. This timer is set when a connection is established. The `tcp_setpersist` function, shown in [Figure 25.22](#), calculates and initializes the persist timer.

**Figure 25.22. `tcp_setpersist` function: calculating and initializing the persist timer**

---

```

493 void
494 tcp_setpersist(tp)
495 struct tcpcb *tp;
496 {
497     t = ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1;
498     if (tp->t_timer[TCPT_REXMT])
499         panic("tcp_output REXMT");
500     /*
501      * Start/restart persistence timer.
502      */
503     TCPT_RANGESET(tp->t_timer[TCPT_PERSIST],
504                   t * tcp_backoff[tp->t_rxtshift],
505                   TCPTV_PERSMIN, TCPTV_PERSMAX);
506     if (tp->t_rxtshift < TCP_MAXRXTSHIFT)
507         tp->t_rxtshift++;
508 }

```

---

*tcp\_output.c*

## Check retransmission timer not enabled

**493-499**

A check is made that the retransmission timer is not enabled if the persist timer is about to be set, since the two timers cannot both be active at the same time. If data is being sent, the other side must be advertising an ACK, but the persist timer is being set only if the advertised sequence number has not been received.

## Calculate RTO

**500-505**

The variable  $t$  is set to the  $RTO$  value that was calculated by the function. The equation being solved is

$$RTO = srtt + 2 \times rttvar$$

which is identical to the formula used at the end of substitution we get

$$RTO = \frac{\frac{t_{\text{srtt}}}{4} + t_{\text{rttvar}}}{2}$$

which is the value computed for the variable t.

## Apply exponential backoff

506-507

An *exponential backoff* is also applied to the RTO by a value from the `tcp_backoff` array:

```
int tcp_backoff[TCP_MAXRXTSHIFT]
{ 1, 2, 4, 8, 16, 32, 64, 64 }
```

When `tcp_output` initially sets the persist timer

```
tp->t_rxtshift = 0;
tcp_setpersist(tp);
```

so the first time `tcp_setpersist` is called, `t_rxtshift` is `tcp_backoff[0]` is 1, t is used as the persist timer. The macro bounds this value between 5 and 60 seconds.

by 1 until it reaches a maximum of 12 (TCP\_M/  
tcp\_backoff[12] is the final entry in the array.

---

## Chapter 25. TCP Timers

---

### 25.10 tcp\_xmit\_timer Function

The next function we look at, `tcp_xmit_timer`, is called when a RTT measurement is collected, to update the smooth mean deviation estimator (*rttvar*).

The argument `rtt` is the RTT measurement to be used in the calculation, using the notation from [Section 25.7](#). It can be

1. **If the timestamp option is present in a segment, then the measured RTT is the current time (tcp\_now). We'll examine the timestamp option in detail later, but for now all we need to know is that `tcp_noack` is set to 1 (Figure 25.8). When a data segment is sent, the sender includes its timestamp, and the other end echoes the timestamp in its acknowledgment it sends back.**
  - If timestamps are not in use and a data segment is sent, then Figure 25.8 shows that the counter `t_rtt` is incremented by the value of `rtt`.

connection. We also mentioned in [Section 25.5](#) so when the acknowledgment is received the counter (in ticks) plus 1.

Typical code in `tcp_input` that calls `tcp_xmit_timer`:

```
if (ts_present)
    tcp_xmit_timer(tp, tcp_now -
else if (tp->t_rtt && SEQ_GT(ti_ack, tp->t_rtseq))
    tcp_xmit_timer(tp, tp->t_rtt + 1);
```

If a timestamp was present in the segment (`ts_present`), it is updated using the current time (`tcp_now`) minus the sequence number of the acknowledgment plus 1. (We describe the reason for adding 1 below.)

If a timestamp is not present, the RTT estimate is updated by acknowledging a data segment that was transmitted. There is one RTT counter per TCP control block (`t_rtt`), so one RTT estimate can be timed per connection. The starting sequence number is stored in `t_rtseq` when the segment is transmitted. An acknowledgment is received that covers that sequence number. If the acknowledgment number (`ti_ack`) is greater than the sequence number of the segment being timed (`t_rtseq`), the RTT is updated as the measured RTT.

Before RFC 1323 timestamps were supported, the RTT was counted by counting clock ticks in `t_rtt`. But this variable

specifies whether a segment is being timed (F than 0, then tcp\_slowtimo adds 1 to it every ! nonzero, it is the number of ticks plus 1. We'll always decrements its second argument by 1 Therefore when timestamps are being used, 1 argument to account for the decrement by 1 i

The greater-than test of the sequence numbers if TCP sends and times a segment with sequence 1), then immediately sends (but can't time) a segment with sequence numbers 1025-2048, and then receives an ACK with ti\_ack for sequence numbers 1-2048 and the ACK ack timed as well as the second (untimed) segment. If timestamps are in use there is no comparison c end sends a timestamp option, it chooses the value TCP to calculate the RTT.

Figure 25.23 shows the first part of the function

**Figure 25.23. `tcp_xmit_timer` function: a smoothed estimate**

---

```

1310 void
1311 tcp_xmit_timer(tp, rtt)
1312 struct tcpcb *tp;
1313 short rtt;
1314 {
1315     short delta;

1316     tcpstat.tcpst_rttupdated++;
1317     if (tp->t_srtt != 0) {
1318         /*
1319          * srtt is stored as fixed point with 3 bits after the
1320          * binary point (i.e., scaled by 8). The following magic
1321          * is equivalent to the smoothing algorithm in rfc793 with
1322          * an alpha of .875 (srtt = rtt/8 + srtt*7/8 in fixed
1323          * point). Adjust rtt to origin 0.
1324         */
1325         delta = rtt - 1 - (tp->t_srtt >> TCP_RTT_SHIFT);
1326         if ((tp->t_srtt += delta) <= 0)
1327             tp->t_srtt = 1;
1328         /*
1329          * We accumulate a smoothed rtt variance (actually, a
1330          * smoothed mean difference), then set the retransmit
1331          * timer to smoothed rtt + 4 times the smoothed variance.
1332          * rttvar is stored as fixed point with 2 bits after the
1333          * binary point (scaled by 4). The following is
1334          * equivalent to rfc793 smoothing with an alpha of .75
1335          * (rttvar = rttvar*3/4 + |delta| / 4). This replaces
1336          * rfc793's wired-in beta.
1337         */
1338         if (delta < 0)
1339             delta = -delta;
1340         delta -= (tp->t_rttvar >> TCP_RTTVAR_SHIFT);
1341         if ((tp->t_rttvar += delta) <= 0)
1342             tp->t_rttvar = 1;
1343     } else {
1344         /*
1345          * No rtt measurement yet - use the unsmoothed rtt.
1346          * Set the variance to half the rtt (so our first
1347          * retransmit happens at 3*rtt).
1348         */
1349         tp->t_srtt = rtt << TCP_RTT_SHIFT;
1350         tp->t_rttvar = rtt << (TCP_RTTVAR_SHIFT - 1);
1351     }

```

---

tcp\_input.c

## Update smoothed estimators

*1310-1325*

Recall that `tcp_newtcpcb` initialized the smooth estimators to zero, indicating that no measurements have been made. The code then updates the smoothed estimator by taking the difference between the measured RTT and the current smoothed estimator, in unscaled ticks. `t_srtt` is divided by `TCP_RTT_SHIFT` and `t_rttvar` is divided by `TCP_RTTVAR_SHIFT`.

1326-1327

The smoothed RTT estimator is updated using t

$$srtt \leftarrow srtt + g \times delta$$

Since the gain  $g$  is  $1/8$ , this equation is

$$8 \times srtt \leftarrow 8 \times srtt + delta$$

which is

$$t\_srtt \leftarrow t\_srtt + delta$$

1328-1342

The mean deviation estimator is updated using

$$rttvar \leftarrow rttvar + h(|delta| - rttvar)$$

Substituting  $1/4$  for  $h$  and the scaled variable  $t$ ,

$$\frac{t\_rttvar}{4} \leftarrow \frac{t\_rttvar}{4} + \frac{|delta| - \frac{t\_rttvar}{4}}{4}$$

which is

$$t_{rttvar} \leftarrow t_{rttvar} + |delta| - \frac{t_{rttvar}}{4}$$

This final equation corresponds to the C code.

## Initialize smoothed estimators on first RTT m

1343-1350

If this is the first RTT measured for this connection, then  $t_{rttvar}$  is initialized to the measured RTT. These calculations are based on the argument  $rtt$ , which we said is the measured RTT. This is the same as the earlier calculation of  $delta$  subtracted 1 from the result.

$$srtt = nticks + 1$$

or

$$\frac{t_{srtt}}{8} = nticks + 1$$

which is

$$t\_srtt = (nticks + 1) \times 8$$

The smoothed mean deviation is set to one-half

$$rttvar = \frac{srtt}{2}$$

which is

$$\frac{t\_rttvar}{4} = \frac{nticks + 1}{2}$$

or

$$t\_rttvar = (nticks + 1) \times 2$$

The comment in the code states that this initial deviation yields an initial *RTO* of  $3 \times srtt$ . Since

$$RTO = srtt + 4 \times rttvar$$

substituting for *rttvar* gives us

$$RTO = srtt + 4 \times \frac{srtt}{2}$$

which is indeed

$$RTO = 3 \times srtt$$

Figure 25.24 shows the final part of the `tcp_xmit_timer` function.

### Figure 25.24. `tcp_xmit_timer` function

```
1352     tp->t_rtt = 0;                               tcp_input.c
1353     tp->t_rxtshift = 0;
1354
1355     /*
1356      * the retransmit should happen at rtt + 4 * rttvar.
1357      * Because of the way we do the smoothing, srtt and rttvar
1358      * will each average +1/2 tick of bias. When we compute
1359      * the retransmit timer, we want 1/2 tick of rounding and
1360      * 1 extra tick because of +/-1/2 tick uncertainty in the
1361      * firing of the timer. The bias will give us exactly the
1362      * 1.5 tick we need. But, because the bias is
1363      * statistical, we have to test that we don't drop below
1364      * the minimum feasible timer (which is 2 ticks).
1365      */
1366     TCPT_RANGESET(tp->t_rxcur, TCP_REXMTVAL(tp),
1367                   tp->t_rttmin, TCPTV_REXMTMAX);
1368
1369     /*
1370      * We received an ack for a packet that wasn't retransmitted;
1371      * it is probably safe to discard any error indications we've
1372      * received recently. This isn't quite right, but close enough
1373      * for now (a route might have failed after we sent a segment,
1374      * and the return path might not be symmetrical).
1375      */
1376     tp->t_softerror = 0;
1377 }
```

1352-1353

The RTT counter (`t_rtt`) and the retransmission counter are reset to 0 in preparation for timing and transmission.

1354-1366

The next *RTO* to use for the connection (*t\_rxcl*

```
#define TCP_REXMTVAL(tp) \
(((tp)->t_srtt >> TCP_RT
```

This is the now-familiar equation

$$RTO = srtt + 4 \times rttvar$$

using the scaled variables updated by *tcp\_xmit* variables for *srtt* and *rttvar*, we have

$$RTO = \frac{t_{srtt}}{8} + 4 \times \frac{t_{rttvar}}{4}$$

$$= \frac{t_{srtt}}{8} + t_{rttvar}$$

which corresponds to the macro. The calculated the minimum *RTO* for this connection (*t\_rttmin* and 128 ticks (*TCPTV\_REXMTMAX*).

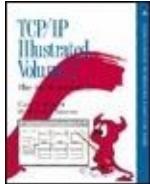
## Clear soft error variable

1367-1374

Since `tcp_xmit_timer` is called only when an ac data segment that was sent, if a soft error was (`t_softerror`), that error is discarded. We describe next section.

---

Team-Fly



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

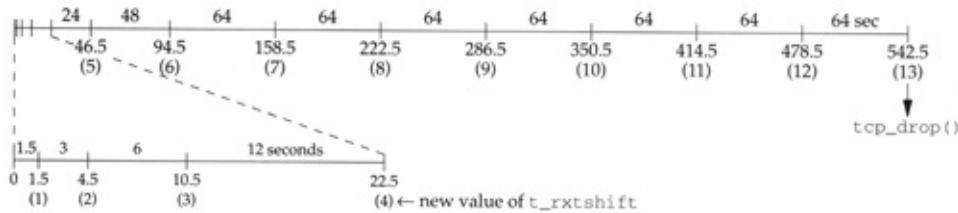
## Chapter 25. TCP Timers

### 25.11 Retransmission Timeout: `tcp_timers` Function

We now return to the `tcp_timers` function and cover the final case that we didn't present in [Section 25.6](#): the one that handles the expiration of the retransmission timer. This code is executed when a data segment that was transmitted has not been acknowledged by the other end within the *RTO*.

[Figure 25.25](#) summarizes the actions caused by the retransmission timer. We assume that the first timeout calculated by `tcp_output` is 1.5 seconds, which is typical for a LAN (see [Figure 21.1](#) of Volume 1).

**Figure 25.25. Summary of retransmission timer when sending data.**



The x-axis is labeled with the time in seconds: 0, 1.5, 4.5, and so on. Below each of these numbers we show the value of `t_rxtshift` that is used in the code we're about to examine. Only after 12 retransmissions and a total of 542.5 seconds (just over 9 minutes) does TCP give up and drop the connection.

RFC 793 recommended that an open of a new connection, active or passive, allow a parameter specifying the total timeout period for data sent by TCP. This is the total amount of time TCP will try to send a given segment before giving up and terminating the connection. The recommended default was 5 minutes.

RFC 1122 requires that an application

must be able to specify a parameter for a connection giving either the total number of retransmissions or the total timeout value for data sent by TCP. This parameter can be specified as "infinity," meaning TCP never gives up, allowing, perhaps, an interactive user the choice of when to give up.

We'll see in the code described shortly that Net/3 does not give the application any of this control: a fixed number of retransmissions (12) always occurs before TCP gives up, and the total timeout before giving up depends on the RTT.

The first half of the retransmission timeout case is shown in [Figure 25.26](#).

**Figure 25.26. `tcp_timers` function:  
expiration of retransmission timer, first  
half.**

---

```

140      /*
141      * Retransmission timer went off. Message has not
142      * been acked within retransmit interval. Back off
143      * to a longer retransmit interval and retransmit one segment.
144      */
145     case TCPT_REXMT:
146         if (++tp->t_rxtshift > TCP_MAXRXTSHIFT) {
147             tp->t_rxtshift = TCP_MAXRXTSHIFT;
148             tcpstat.tcpst_timeoutdrop++;
149             tp = tcp_drop(tp, tp->t_softerror ?
150                           tp->t_softerror : ETIMEDOUT);
151             break;
152         }
153         tcpstat.tcpst_rexmttimeo++;
154         rexmt = TCP_REXMTVAL(tp) * tcp_backoff(tp->t_rxtshift);
155         TCPT_RANGESET(tp->t_rxtcur, rexmt,
156                       tp->t_rttmin, TCPTV_REXMTMAX);
157         tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
158         /*
159         * If losing, let the lower level know and try for
160         * a better route. Also, if we backed off this far,
161         * our srtt estimate is probably bogus. Clobber it
162         * so we'll take the next rtt measurement as our srtt;
163         * move the current srtt into rttvar to keep the current
164         * retransmit times until then.
165         */
166         if (tp->t_rxtshift > TCP_MAXRXTSHIFT / 4) {
167             in_losing(tp->t_inpcb);
168             tp->t_rttvar += (tp->t_srtt >> TCP_RTT_SHIFT);
169             tp->t_srtt = 0;
170         }
171         tp->snd_nxt = tp->snd_una;
172         /*
173         * If timing a segment in this window, stop the timer.
174         */
175         tp->t_rtt = 0;

```

---

tcp\_timer.c

## Increment shift count

**146**

The retransmission shift count (`t_rxtshift`) is incremented, and if the value exceeds 12 (`TCP_MAXRXTSHIFT`) it is time to drop the connection. This new value of `t_rxtshift` is what we show in [Figure 25.25](#). Notice the difference between this dropping of a connection because an

acknowledgment is not received from the other end in response to data sent by TCP, and the keepalive timer, which drops a connection after a long period of inactivity and no response from the other end. Both report the error ETIMEDOUT to the process, unless a soft error is received for the connection.

## Drop connection

147-152

A *soft error* is one that doesn't cause TCP to terminate an established connection or an attempt to establish a connection, but the soft error is recorded in case TCP gives up later. For example, if TCP retransmits a SYN segment to establish a connection, receiving nothing in response, the error returned to the process will be ETIMEDOUT. But if during the retransmissions an ICMP host unreachable is received for the connection, that is considered a soft error and stored in t\_softerror by tcp\_notify. If TCP finally gives up the retransmissions, the error

returned to the process will be EHOSTUNREACH instead of ETIMEDOUT, providing more information to the process. If TCP receives an RST on the connection in response to the SYN, that's considered a *hard error* and the connection is terminated immediately with an error of ECONNREFUSED ([Figure 28.18](#)).

## Calculate new RTO

153-157

The next *RTO* is calculated using the TCP\_REXMTVAL macro, applying an exponential backoff. In this code, t\_rxtshift will be 1 the first time a given segment is retransmitted, so the *RTO* will be twice the value calculated by TCP\_REXMTVAL. This value is stored in t\_rxtcur and as the retransmission timer for the connection, t\_timer[TCPT\_REXMT]. The value stored in t\_rxtcur is used in tcp\_input when the retransmission timer is restarted ([Figures 28.12 and 29.6](#)).

## Ask IP to find a new route

158-167

If this segment has been retransmitted four or more times, `in_losing` releases the cached route (if there is one), so when the segment is retransmitted by `tcp_output` (at the end of this case statement in [Figure 25.27](#)) a new, and hopefully better, route will be chosen. In [Figure 25.25](#) `in_losing` is called each time the retransmission timer expires, starting with the retransmission at time 22.5.

**Figure 25.27. `tcp_timers` function:  
expiration of retransmission timer,  
second half.**

---

```

176     /*
177      * Close the congestion window down to one segment
178      * (we'll open it by one segment for each ack we get).
179      * Since we probably have a window's worth of unacked
180      * data accumulated, this "slow start" keeps us from
181      * dumping all that data as back-to-back packets (which
182      * might overwhelm an intermediate gateway).
183      *
184      * There are two phases to the opening: Initially we
185      * open by one mss on each ack. This makes the window
186      * size increase exponentially with time. If the
187      * window is larger than the path can handle, this
188      * exponential growth results in dropped packet(s)
189      * almost immediately. To get more time between
190      * drops but still "push" the network to take advantage
191      * of improving conditions, we switch from exponential
192      * to linear window opening at some threshold size.
193      * For a threshold, we use half the current window
194      * size, truncated to a multiple of the mss.
195      *
196      * (the minimum cwnd that will give us exponential
197      * growth is 2 mss. We don't allow the threshold
198      * to go below this.)
199      */
200 {
201     u_int    win = min(tp->snd_wnd, tp->snd_cwnd) / 2 / tp->t_maxseg;
202     if (win < 2)
203         win = 2;
204     tp->snd_cwnd = tp->t_maxseg;
205     tp->snd_ssthresh = win * tp->t_maxseg;
206     tp->t_dupacks = 0;
207 }
208 (void) tcp_output(tp);
209 .break;

```

---

tcp\_timer.c

## Clear estimators

168-170

The smoothed RTT estimator (`t_srtt`) is set to 0, which is what `t_newtcpcb` did. This forces `tcp_xmit_timer` to use the next measured RTT as the smoothed RTT estimator. This is done because the retransmitted segment has been sent four or more times, implying that TCP's smoothed RTT estimator is probably way

off. But if the retransmission timer expires again, at the beginning of this case statement the *RTO* is calculated by `TCP_REXMTVAL`. That calculation should generate the same value as it did for this retransmission (which will then be exponentially backed off), even though `t_srtt` is set to 0. (The retransmission at time 42.464 in [Figure 25.28](#) is an example of what's happening here.)

**Figure 25.28. Values of RTT variables and estimators during example.**

To accomplish this the value of  $t_{rttvar}$  is changed as follows. The next time the  $RTO$  is calculated, the equation

$$RTO = \frac{t\_srtt}{8} + t\_rttvar$$

is evaluated. Since  $t_{\text{srtt}}$  will be 0, if  $t_{\text{rttvar}}$  is increased by  $t_{\text{srtt}}$  divided by 8,  $RTO$  will have the same value. If the retransmission timer expires again for this segment (e.g., times 84.064 through 217.184 in [Figure 25.28](#)), when this code is executed again  $t_{\text{srtt}}$  will be 0, so  $t_{\text{rttvar}}$  won't change.

## Force retransmission of oldest unacknowledged data

171

The next send sequence number ( $\text{snd\_nxt}$ ) is set to the oldest unacknowledged sequence number ( $\text{snd\_una}$ ). Recall from [Figure 24.17](#) that  $\text{snd\_nxt}$  can be greater than  $\text{snd\_una}$ . By moving  $\text{snd\_nxt}$  back, the retransmission will be the oldest segment that hasn't been acknowledged.

## Karn's algorithm

172-175

The RTT counter,  $t_{rtt}$ , is set to 0, in case the last segment transmitted was being timed. Karn's algorithm says that even if an ACK of that segment is received, since the segment is about to be retransmitted, any timing of the segment is worthless since the ACK could be for the first transmission or for the retransmission. The algorithm is described in [Karn and Partridge 1987] and in Section 21.3 of Volume 1. Therefore the only segments that are timed using the  $t_{rtt}$  counter and used to update the RTT estimators are those that are not retransmitted. We'll see in Figure 29.6 that the use of RFC 1323 timestamps overrides Karn's algorithm.

## Slow Start and Congestion Avoidance

The second half of this case is shown in Figure 25.27. It performs slow start and congestion avoidance and retransmits the oldest unacknowledged segment.

Since a retransmission timeout has occurred, this is a strong indication of congestion in the network. TCP's

*congestion avoidance algorithm* comes into play, and when a segment is eventually acknowledged by the other end, TCP's *slow start* algorithm will continue the data transmission on the connection at a slower rate. Sections 20.6 and 21.6 of Volume 1 describe the two algorithms in detail.

176-205

win is set to one-half of the current window size (the minimum of the receiver's advertised window, `snd_wnd`, and the sender's congestion window, `snd_cwnd`) in segments, not bytes (hence the division by `t_maxseg`). Its minimum value is two segments. This records one-half of the window size when the congestion occurred, assuming one cause of the congestion is our sending segments too rapidly into the network. This becomes the slow start threshold, `t_ssthresh` (which is stored in bytes, hence the multiplication by `t_maxseg`). The congestion window, `snd_cwnd`, is set to one segment, which forces slow start.

This code is enclosed in braces because

it was added between the 4.3BSD and Net/1 releases and required its own local variable (`win`).

206

The counter of consecutive duplicate ACKs, `t_dupacks` (which is used by the fast retransmit algorithm in [Section 29.4](#)), is set to 0. We'll see how this counter is used with TCP's fast retransmit and fast recovery algorithms in [Chapter 29](#).

208

`tcp_output` resends a segment containing the oldest unacknowledged sequence number. This is the retransmission caused by the retransmission timer expiring.

## Accuracy

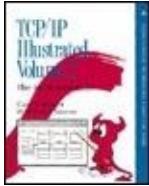
How accurate are these estimators that TCP maintains? At first they appear too coarse, since the RTTs are measured in multiples of 500 ms. The mean and mean deviation are maintained with additional accuracy (factors of 8 and 4 respectively),

but LANs have RTTs on the order of milliseconds, and a transcontinental RTT is around 60 ms. What these estimators provide is a solid upper bound on the RTT so that the retransmission timeout can be set without worrying that the timeout is too small, causing unnecessary and wasteful retransmissions.

[Brakmo, O'Malley, and Peterson 1994] describe a TCP implementation that provides higher-resolution RTT measurements. This is done by recording the system clock (which has a much higher resolution than 500 ms) when a segment is transmitted and reading the system clock when the ACK is received, calculating a higher-resolution RTT.

The timestamp option provided by Net/3 ([Section 26.6](#)) can provide higher-resolution RTTs, but Net/3 sets the resolution of these timestamps to 500 ms.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.12 An RTT Example

We now go through an actual example to see how the calculations are performed. We transfer 12288 bytes from the host bsdi to vangogh.cs.berkeley.edu. During the transfer we purposely bring down the PPP link being used and then bring it back up, to see how timeouts and retransmissions are handled. To transfer the data we use our sock program (described in Appendix C of Volume 1) with the -D option, to enable the SO\_DEBUG socket option ([Section 27.10](#)). After the transfer is complete we examine the debug records left in the kernel's circular buffer using the trpt(8) program and print the desired timer variables from

the TCP control block.

[Figure 25.28](#) shows the calculations that occur at the various times. We use the notation  $M:N$  to mean that sequence numbers  $M$  through and including  $N - 1$  are sent. Each segment in this example contains 512 bytes. The notation "ack  $M$ " means that the acknowledgment field of the ACK is  $M$ . The column labeled "actual delta (ms)" shows the time difference between the RTT timer going on and going off. The column labeled "rtt (arg.)" shows the second argument to the `tcp_xmit_timer` function: the number of clock ticks plus 1 between the RTT timer going on and going off.

The function `tcp_newtcpcb` initializes `t_srtt`, `t_rttvar`, and `t_rxtcur` to the values shown at time 0.0.

The first segment timed is the initial SYN. When its ACK is received 365 ms later, `tcp_xmit_timer` is called with an `rtt` argument of 2. Since this is the first RTT measurement (`t_srtt` is 0), the `else` clause in [Figure 25.23](#) calculates the first values

of the smoothed estimators.

The data segment containing bytes 1 through 512 is the next segment timed, and the RTT variables are updated at time 1.259 when its ACK is received.

The next three segments show how ACKs are cumulative. The timer is started at time 1.260 when bytes 513 through 1024 are sent. Another segment is sent with bytes 1025 through 1536, and the ACK received at time 2.206 acknowledges both data segments. The RTT estimators are then updated, since the ACK covers the starting sequence number being timed (513).

The segment with bytes 1537 through 2048 is transmitted at time 2.206 and the timer is started. Just that segment is acknowledged at time 3.132, and the estimators updated.

The data segment at time 3.132 is timed and the retransmission timer is set to 5 ticks (the current value of `t_rxtcur`). Somewhere around this time the PPP link

between the routers sun and netb is taken down and then brought back up, a procedure that takes a few minutes. When the retransmission timer expires at time 6.064, the code in [Figure 25.26](#) is executed to update the RTT variables.

`t_rxtshift` is incremented from 0 to 1 and `t_rxtcur` is set to 10 ticks (the exponential backoff). A segment starting with the oldest unacknowledged sequence number (`snd_una`, which is 3073) is retransmitted. After 5 seconds the timer expires again, `t_rxtshift` is incremented to 2, and the retransmission timer is set to 20 ticks.

When the retransmission timer expires at time 42.464, `t_srtt` is set to 0 and `t_rttvar` is set to 5. As we mentioned in our discussion of [Figure 25.26](#), this leaves the calculation of `t_rxtcur` the same (so the next calculation yields 160), but by setting `t_srtt` to 0, the next time the RTT estimators are updated (at time 218.834), the measured RTT becomes the smoothed RTT, as if the connection were starting fresh.

The rest of the data transfer continues,

and the estimators are updated a few more times.

---

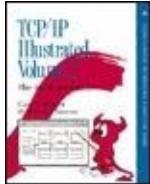
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 25. TCP Timers

### 25.13 Summary

The two functions `tcp_fasttimo` and `tcp_slowtimo` are called by the kernel every 200 ms and every 500 ms, respectively. These two functions drive TCP's per-connection timer maintenance.

TCP maintains the following seven timers for each connection:

- a connection-establishment timer,
- a retransmission timer,
- a delayed ACK timer,
- a persist timer,

- a keepalive timer,
- a FIN\_WAIT\_2 timer, and
- a 2MSL timer.

The delayed ACK timer is different from the other six, since when it is set it means a delayed ACK must be sent the next time TCP's 200-ms timer expires. The other six timers are counters that are decremented by 1 every time TCP's 500-ms timer expires. When any one of the counters reaches 0, the appropriate action is taken: drop the connection, retransmit a segment, send a keepalive probe, and so on, as described in this chapter. Since some of the timers are mutually exclusive, the six timers are really implemented using four counters, which complicates the code.

This chapter also introduced the recommended way to calculate values for the retransmission timer. TCP maintains two smoothed estimators for a connection: the round-trip time and the mean deviation of the RTT. Although the

algorithms are simple and elegant, these estimators are maintained as scaled fixed-point numbers (to provide adequate precision without using floating-point code within the kernel), which complicates the code.

## Exercises

**25.1** How efficient is TCP's fast timeout function? (*Hint:* Look at the number of delayed ACKs in [Figure 24.5](#).) Suggest alternative implementations.

**25.2** Why do you think the initialization of `tcp_maxidle` is in the `tcp_slowtimo` function instead of the `tcp_init` function?

**25.3** `tcp_slowtimo` increments `t_idle`, which we said counts the clock ticks since a segment was last received on the connection. Should TCP also count the idle time since a segment

was last sent on a connection?

Rewrite the code in [Figure 25.10](#) to  
**25.4** separate the logic for the two  
different uses of the TCPT\_2MSL  
counter.

75 seconds after the connection in  
**25.5** [Figure 25.12](#) enters the FIN\_WAIT\_2  
state a duplicate ACK is received on  
the connection. What happens?

A connection has been idle for 1 hour  
when the application sets the  
**25.6** SO\_KEEPALIVE option. Will the first  
keepalive probe be sent 1 or 2 hours  
in the future?

**25.7** Why is `tcp_rttdflt` a global variable  
and not a constant?

Rewrite the code related to [Exercise 25.6](#)  
**25.8** to implement the alternate  
behavior.

---

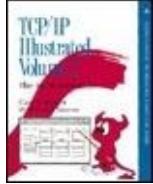
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 26. TCP Output

[Section 26.1. Introduction](#)

[Section 26.2. tcp\\_output Overview](#)

[Section 26.3. Determine if a Segment Should be Sent](#)

[Section 26.4. TCP Options](#)

[Section 26.5. Window Scale Option](#)

[Section 26.6. Timestamp Option](#)

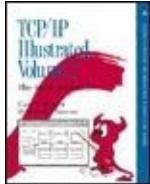
[Section 26.7. Send a Segment](#)

[Section 26.8. tcp\\_template Function](#)

[Section 26.9. tcp\\_respond Function](#)

[Section 26.10. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

### 26.1 Introduction

The function `tcp_output` is called whenever a segment needs to be sent on a connection. There are numerous calls to this function from other TCP functions:

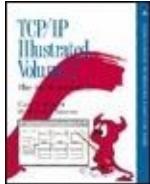
- `tcp_usrreq` calls it for various requests: `PRU_CONNECT` to send the initial SYN, `PRU_SHUTDOWN` to send a FIN, `PRU_RCVD` in case a window update can be sent after the process has read some data from the socket receive buffer, `PRU_SEND` to send data, and `PRU_SENDOOB` to send out-of-band data.
- `tcp_fasttimo` calls it to send a delayed

ACK.

- `tcp_timers` calls it to retransmit a segment when the retransmission timer expires.
- `tcp_timers` calls it to send a persist probe when the persist timer expires.
- `tcp_drop` calls it to send an RST.
- `tcp_disconnect` calls it to send a FIN.
- `tcp_input` calls it when output is required or when an immediate ACK should be sent.
- `tcp_input` calls it when a pure ACK is processed by the header prediction code and there is more data to send. (*A pure ACK* is a segment without data that just acknowledges data.)
- `tcp_input` calls it when the third consecutive duplicate ACK is received, to send a single segment (the fast retransmit algorithm).

`tcp_output` first determines whether a

segment should be sent or not. TCP output is controlled by numerous factors other than data being ready to send to the other end of the connection. For example, the other end might be advertising a window of size 0 that stops TCP from sending anything, the Nagle algorithm prevents TCP from sending lots of small segments, and slow start and congestion avoidance limit the amount of data TCP can send on a connection. Conversely, some functions set flags just to force `tcp_output` to send a segment, such as the `TF_ACKNOW` flag that means an ACK should be sent immediately and not delayed. If `tcp_output` decides not to send a segment, the data (if any) is left in the socket's send buffer for a later call to this function.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

## 26.2 tcp\_output Overview

tcp\_output is a large function, so we'll discuss it in 14 parts. [Figure 26.1](#) shows the outline of the function.

**Figure 26.1. tcp\_output function: overview.**

---

```

43 int
44 tcp_output(tp)
45 struct tcpcb *tp;
46 {
47     struct socket *so = tp->t_inpcb->inp_socket;
48     long    len, win;
49     int     off, flags, error;
50     struct mbuf *m;
51     struct tcphdr *ti;
52     u_char  opt[MAX_TCPOPTLEN];
53     unsigned optlen, hdrlen;
54     int     idle, sendalot;

55     /*
56      * Determine length of data that should be transmitted
57      * and flags that will be used.
58      * If there are some data or critical controls (SYN, RST)
59      * to send, then transmit; otherwise, investigate further.
60      */
61     idle = (tp->snd_max == tp->snd_una);
62     if (idle && tp->t_idle >= tp->t_rxtcur)
63     /*
64      * We have been idle for "a while" and no acks are
65      * expected to clock out any data we send --
66      * slow start to get ack "clock" running again.
67      */
68     tp->snd_cwnd = tp->t_maxseg;

69 again:
70     sendalot = 0; /* set nonzero if more than one segment to output */

    /* look for a reason to send a segment; */
    /* goto send if a segment should be sent */

218 /*
219  * No reason to send a segment, just return.
220 */
221 return (0);

222 send:

    /* form output segment, call ip_output() */

489 if (sendalot)
490     goto again;
491 return (0);

```

---

tcp\_output.c

## Is an ACK expected from the other end?

61

idle is true if the maximum sequence number sent (snd\_max) equals the oldest

unacknowledged sequence number (snd\_una), that is, if an ACK is not expected from the other end. In [Figure 24.17](#) idle would be 0, since an ACK is expected for sequence numbers 46, which have been sent but not yet acknowledged.

## Go back to slow start

62-68

If an ACK is not expected from the other end and a segment has not been received from the other end in one RTO, the congestion window is set to one segment ( $t_{\text{maxseg}}$  bytes). This forces slow start to occur for this connection the next time a segment is sent. When a significant pause occurs in the data transmission ("significant" being more than the RTT), the network conditions can change from what was previously measured on the connection. Net/3 assumes the worst and returns to slow start.

## Send more than one segment

69-70

When send is jumped to, a single segment is sent by calling ip\_output. But if tcp\_output determines that more than one segment can be sent, sendalot is set to 1, and the function tries to send another segment. Therefore, one call to tcp\_output can result in multiple segments being sent.

## Chapter 26. TCP Output

### 26.3 Determine if a Segment Should Be Sent

Sometimes `tcp_output` is called but a segment is not sent. This happens when the PRU\_RCVD request is generated when the socket has data available from the socket's receive buffer, passing the data to the process that removed enough data that TCI was updated. The other end with a new window advertisement is not a certainty. The first half of `tcp_output` determines whether or not to send a segment to the other end. If not, the function returns without sending a segment.

Figure 26.2 shows the first of the tests to determine whether a segment should be sent.

**Figure 26.2. `tcp_output` function: determining whether a segment should be sent.**

---

```

71     off = tp->snd_nxt - tp->snd_una;
72     win = min(tp->snd_wnd, tp->snd_cwnd);
73     flags = tcp_outflags[tp->t_state];
74     /*
75      * If in persist timeout with window of 0, send 1 byte.
76      * Otherwise, if window is small but nonzero
77      * and timer expired, we will send what we can
78      * and go to transmit state.
79      */
80     if (tp->t_force) {
81         if (win == 0) {
82             /*
83              * If we still have some data to send, then
84              * clear the FIN bit. Usually this would
85              * happen below when it realizes that we
86              * aren't sending all the data. However,
87              * if we have exactly 1 byte of unsent data,
88              * then it won't clear the FIN bit below,
89              * and if we are in persist state, we wind
90              * up sending the packet without recording
91              * that we sent the FIN bit.
92              *
93              * We can't just blindly clear the FIN bit,
94              * because if we don't have any more data
95              * to send then the probe will be the FIN
96              * itself.
97              */
98             if (off < so->so_snd.sb_cc)
99                 flags &= ~TH_FIN;
100            win = 1;
101        } else {
102            tp->t_timer[TCPT_PERSIST] = 0;
103            tp->t_rxtshift = 0;
104        }
105    }

```

---

tcp\_output.c

**71-72**

off is the offset in bytes from the beginning of the data byte to send. The first off bytes in the sequence, snd\_una, have already been sent and are waiting.

win is the minimum of the window advertised by the congestion window (snd\_cwnd).

**73**

The tcp\_outflags array was shown in [Figure 24](#). The value that is fetched and stored in flags depends on the state of the connection.

connection. flags contains the combination of tl and TH\_SYN flag bits to send to the other end. TH\_PUSH and TH\_URG, will be logically ORed if the segment is sent.

## 74-105

The flag t\_force is set nonzero when the persist band data is being sent. These two conditions i

```
tp->t_force = 1;  
error = tcp_output(tp);  
tp->t_force = 0;
```

This forces TCP to send a segment when it norr

If win is 0, the connection is in the persist state. The FIN flag is cleared if there is more data in the segment. t\_force must be set to 1 byte to force out a single byte

If win is nonzero, out-of-band data is being sent. The FIN flag is cleared and the exponential backoff index, t\_rx

[Figure 26.3](#) shows the next part of tcp\_output, calculating the data to send.

**Figure 26.3. tcp\_output function: calculate the data to send.**

```

106     len = min(so->so_snd.sb_cc, win) - off;
107     if (len < 0) {
108         /*
109          * If FIN has been sent but not acked,
110          * but we haven't been called to retransmit,
111          * len will be -1. Otherwise, window shrank
112          * after we sent into it. If window shrank to 0,
113          * cancel pending retransmit and pull snd_nxt
114          * back to (closed) window. We will enter persist
115          * state below. If the window didn't close completely,
116          * just wait for an ACK.
117         */
118         len = 0;
119         if (win == 0) {
120             tp->t_timer[TCPT_RXTMR] = 0;
121             tp->snd_nxt = tp->snd_una;
122         }
123     }
124     if (len > tp->t_maxseg) {
125         len = tp->t_maxseg;
126         sendalot = 1;
127     }
128     if (SEQ_LT(tp->snd_nxt + len, tp->snd_una + so->so_snd.sb_cc))
129         flags &= ~TH_FIN;
130     win = sbspace(&so->so_rcv);

```

tcp\_output.c

## Calculate amount of data to send

**106**

len is the minimum of the number of bytes in tl  
 is the minimum of the receiver's advertised wir  
 window, perhaps 1 byte if output is being force  
 that many bytes at the beginning of the send b  
 and are awaiting acknowledgment.

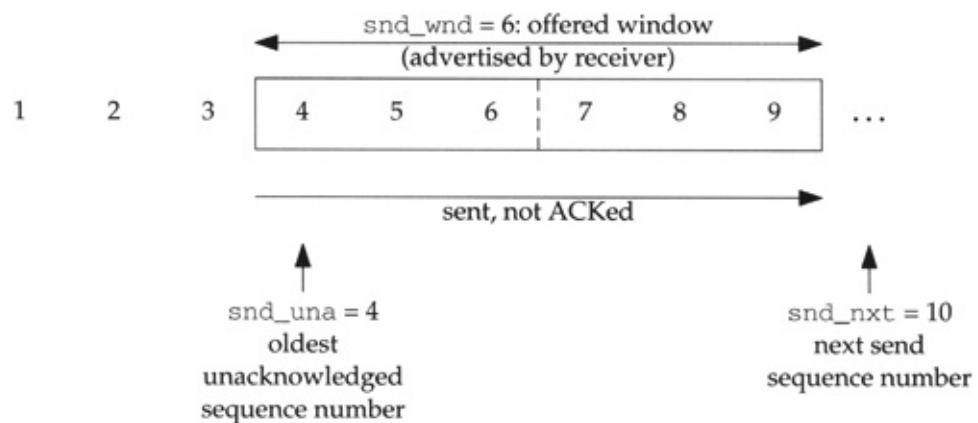
## Check for window shrink

**107-117**

One way for len to be less than 0 occurs if the i

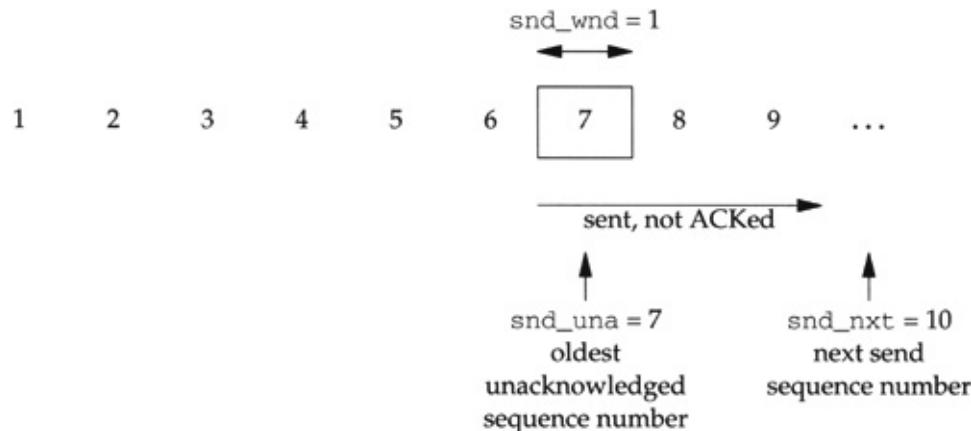
that is, the receiver moves the right edge of the window. The following example demonstrates how this can happen. Suppose the receiver advertises a window of 6 bytes and TCP transmits sequence numbers 4, 5, and 6. TCP immediately transmits another segment with sequence number 7. Figure 26.4 shows the status of our end after transmission.

**Figure 26.4. Send buffer after bytes 4–6 have been transmitted.**



Then an ACK is received with an acknowledgment number of 7, meaning all data up through and including byte 6) but was not acknowledged. The receiver has shrunk the window, as shown in Figure 26.5.

**Figure 26.5. Send buffer after receiving an ACK for bytes 4 through 6.**



Performing the calculations in [Figures 26.2](#) and shrunk, we have

```

off = snd_nxt - snd_una = 10 - 7
win = 1
len = min(so_snd.sb_cc, win) - c

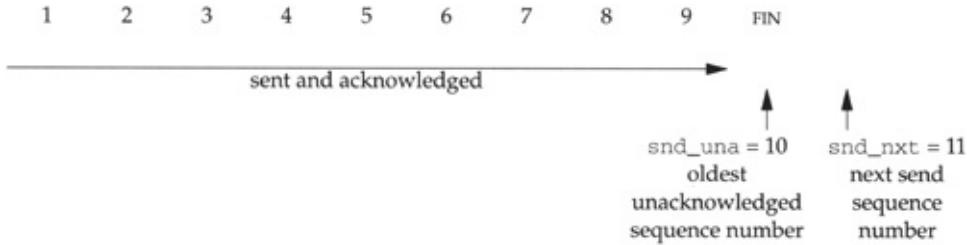
```

assuming the send buffer contains only bytes 7

Both RFC 793 and RFC 1122 strongly discourage this. Nevertheless, implementations must be prepared for scenarios such as this comes under the *Robustness Principle* mentioned in RFC 791: "Be liberal in what you accept, be strict in what you send."

Another way for len to be less than 0 occurs if the byte at index 7 is acknowledged and not retransmitted. (See [Exercise 26.6](#).)

## Figure 26.6. Bytes 1 through 9 have been sent and acknowledged, then connection is closed



This figure continues [Figure 26.4](#), assuming that byte 8, and 9 is acknowledged, which sets `snd_una` to 10. Closing the connection, causing the FIN to be sent. We can calculate the offset when the FIN is sent, `snd_nxt` is incremented by 1 (the sequence number of the FIN), which in this example sets `off` to 11. Performing the calculation shown in [Figure 26.3](#), we have

```
off = snd_nxt - snd_una = 11 - 1  
win = 6  
len = min(so_snd.sb_cc, win) - c
```

We assume that the receiver advertises a window of size 6, a difference, since the number of bytes in the segment is 11.

**Enter persist state**

len is set to 0. If the advertised window is 0, ar canceled by setting the retransmission timer to the left of the window by setting it to the value will enter the persist state later in this function, opens its window, TCP starts retransmitting fro

## **Send one segment at a time**

124-127

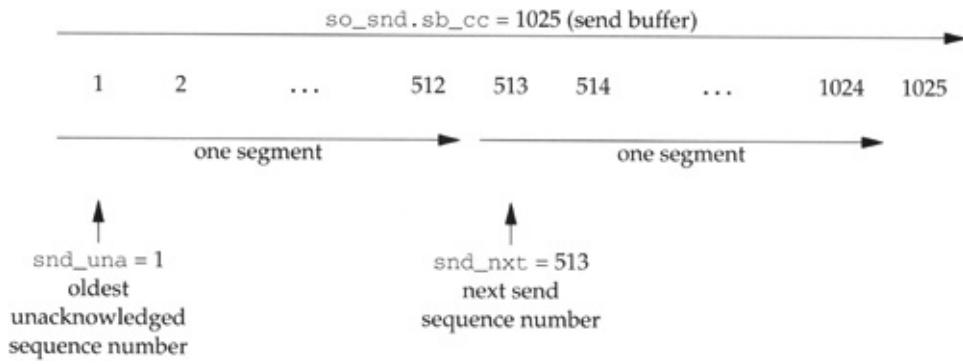
If the amount of data to send exceeds one segi segment and the sendalot flag is set to 1. As s1 causes another loop through tcp\_output after t

## **Turn off FIN flag if send buffer not emptied**

128-129

If the send buffer is not being emptied by this must be cleared (in case it is set in flags). Figure this.

**Figure 26.7. Example of send buffer not bei**



In this example the first 512-byte segment has waiting to be acknowledged) and TCP is about 1 segment (bytes 5121024). There is still 1 byte 1025) and the process closes the connection. Ie and the C expression becomes

SEQ\_LT(1025, 1026)

which is true, so the FIN flag is cleared. If the F on, TCP couldn't send byte 1025 to the receiver

## Calculate window advertisement

130

win is set to the amount of space available in the buffer. This becomes TCP's window advertisement to the other side. This is the second use of this variable in this function. It represents the maximum amount of data TCP could send, but it is not the function it contains the receive window advertisement for the connection.

The silly window syndrome (called *SWS* and de Volume 1) occurs when small amounts of data, segments, are exchanged across a connection. who advertises small windows and by a sender segments. Correct avoidance of the silly window performed by both the sender and the receiver. window avoidance by the sender.

## Figure 26.8. `tcp_output` function: sender

```
131  /*
132   * Sender silly window avoidance.  If connection is idle
133   * and can send all data, a maximum segment,
134   * at least a maximum default-sized segment do it,
135   * or are forced, do it; otherwise don't bother.
136   * If peer's buffer is tiny, then send
137   * when window is at least half open.
138   * If retransmitting (possibly after persist timer forced us
139   * to send into a small window), then must resend.
140   */
141 if (len) {
142     if (len == tp->t_maxseg)
143         goto send;
144     if ((idle || tp->t_flags & TF_NODELAY) &&
145         len + off >= so->so_snd.sb_cc)
146         goto send;
147     if (tp->t_force)
148         goto send;
149     if (len >= tp->max_sndwnd / 2)
150         goto send;
151     if (SEQ_LT(tp->snd_nxt, tp->snd_max))
152         goto send;
153 }
```

*tcp\_output.c*

## Sender silly window avoidance

142-143

If a full-sized segment can be sent, it is sent.

**144-146**

If an ACK is not expected (idle is true), or if the (TF\_NODELAY is true) *and* TCP is emptying the The Nagle algorithm (Section 19.4 of Volume 1 less than a full-sized segment when an ACK is received. This can be disabled using the TCP\_NODELAY socket option. If this is done during an interactive connection (e.g., Telnet or Rlogin), it is likely to cause problems. If no data, this if statement is false, since the Nagle algorithm is the default.

**147-148**

If output is being forced by either the persist timer or the linger option, some data is sent.

**149-150**

If the receiver's window is at least half open, data is sent. It is possible to have peers that always advertise tiny windows, perhaps as a way of conserving bandwidth. The variable max\_sndwnd is calculated by taking the minimum of the current window advertisement ever advertised by the peer and twice the size of the receive buffer. The code then guesses the size of the other end's receive buffer by dividing the maximum window size by two. This guess is used to calculate the size of the receive buffer. The code then never reduces the size of its receive buffer.

**151-152**

If the retransmission timer expired, then a segment is sent. The sequence number is the highest sequence number that has been

[25.26](#) that when the retransmission timer expires, that is, `snd_nxt` is moved to the left edge of the `snd_max`.

The next portion of `tcp_output`, shown in [Figure 26.9](#), send a segment just to advertise a new window called a *window update*.

**Figure 26.9. `tcp_output` function: check if a window update is sent.**

```
154  /*
155   * Compare available window to amount of window
156   * known to peer (as advertised window less
157   * next expected input). If the difference is at least two
158   * max size segments, or at least 50% of the maximum possible
159   * window, then want to send a window update to peer.
160   */
161 if (win > 0) {
162     /*
163      * "adv" is the amount we can increase the window,
164      * taking into account that we are limited by
165      * TCP_MAXWIN << tp->rcv_scale.
166      */
167     long    adv = min(win, (long)TCP_MAXWIN << tp->rcv_scale) -
168             (tp->rcv_adv - tp->rcv_nxt);
169     if (adv >= (long)(2 * tp->t_maxseg))
170         goto send;
171     if (2 * adv >= (long) so->so_rcv.sb_hiwat)
172         goto send;
173 }
```

**154-168**

The expression

`min(win, (long)TCP_MAXWIN << tp-`

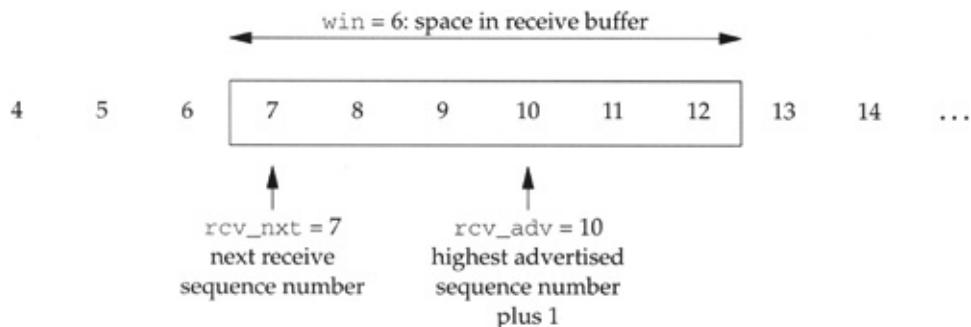
is the smaller of the amount of available space (win) and the maximum size of the window alloc is the maximum window TCP can currently advertise expression

$$(tp->rcv\_adv - tp->rcv\_nxt)$$

is the number of bytes remaining in the last window sent to the other end. Subtracting this from the win shows the number of bytes by which the window has been incremented by `tcp_input` when data is received or incremented by `tcp_output` in [Figure 26.32](#) when the window moves to the right.

Consider [Figure 24.18](#) and assume that a segment is received and that these three bytes are passed to the receiver. This shows the state of the receive space at this point.

**Figure 26.10. Transition from Figure 24.18 to this point received.**



The value of adv is 3, since there are 3 more bytes (bytes 10, 11, and 12) for the other end to fill.

## 169-170

If the window has opened by two or more segments sent. When data is received as full-sized segments, the other received segment to be acknowledged: This is a property. (We show an example of this shortly.)

## 171-172

If the window has opened by at least 50% of the socket's receive buffer high-water mark), it

The next part of `tcp_output`, shown in Figure 26.11, handles flags require TCP to send a segment.

**Figure 26.11. `tcp_output` function: should a segment be sent?**

```
174  /*
175   * Send if we owe peer an ACK.
176   */
177  if (tp->t_flags & TF_ACKNOW)
178      goto send;
179  if (flags & (TH_SYN | TH_RST))
180      goto send;
181  if (SEQ_GT(tp->snd_up, tp->snd_una))
182      goto send;
183  /*
184   * If our state indicates that FIN should be sent
185   * and we have not yet done so, or we're retransmitting the FIN,
186   * then we need to send.
187   */
188  if (flags & TH_FIN &&
189      ((tp->t_flags & TF_SENDFIN) == 0 || tp->snd_nxt == tp->snd_una))
190      goto send;
```

*tcp\_output.c*

174-178

If an immediate ACK is required, a segment is set by various functions: when the 200-ms delay a segment is received out of order (for the fast SYN is received during the three-way handshake), when a FIN is received.

179-180

If flags specifies that a SYN or RST should be sent.

181-182

If the urgent pointer, snd\_up, is beyond the start of the segment is sent. The urgent pointer is set by the code in Figure 30.9.

183-190

If flags specifies that a FIN should be sent, a segment has not already been sent, or if the FIN is being sent, TF\_SENDFIN is set later in this function when the segment is sent.

At this point in tcp\_output there is no need to send the segment. Figure 26.12 shows the final piece of code before tcp\_output returns.

**Figure 26.12. tcp\_output function:**

---

```

191  /*
192   * TCP window updates are not reliable, rather a polling protocol
193   * using 'persist' packets is used to ensure receipt of window
194   * updates. The three 'states' for the output side are:
195   *   idle           not doing retransmits or persists
196   *   persisting     to move a small or zero window
197   *   (re)transmitting and thereby not persisting
198   *
199   * tp->t_timer[TCPT_PERSIST]
200   *      is set when we are in persist state.
201   * tp->t_force
202   *      is set when we are called to send a persist packet.
203   * tp->t_timer[TCPT_REXMT]
204   *      is set when we are retransmitting
205   * The output side is idle when both timers are zero.
206   *
207   * If send window is too small, there is data to transmit, and no
208   * retransmit or persist is pending, then go to persist state.
209   * If nothing happens soon, send when timer expires:
210   * if window is nonzero, transmit what we can,
211   * otherwise force out a byte.
212   */
213 if (so->so_snd.sb_cc && tp->t_timer[TCPT_REXMT] == 0 &&
214     tp->t_timer[TCPT_PERSIST] == 0) {
215     tp->t_rxtshift = 0;
216     tcp_setpersist(tp);
217 }
218 /*
219  * No reason to send a segment, just return.
220  */
221 return (0);

```

---

tcp\_output.c

## 191-217

If there is data in the send buffer to send (`so_s` is non-zero), and both the retransmission timer and the persist timer are zero, then the function returns. This scenario happens when the window advertisement is too small to receive a full-sized segment, and the application has no reason to send a segment.

## 218-221

`tcp_output` returns, since there is no reason to

## Example

A process writes 100 bytes, followed by a write connection. Assume a segment size of 512 bytes. The code in [Figure 26.8](#) (lines 144–146) sends a data since the connection is idle and TCP is empty.

When 50-byte write occurs, the code in [Figure 26.8](#) sends a segment: the amount of data is not a full-sized segment, but it is not idle (assume TCP is awaiting the ACK for the previous segment). Since the Nagle algorithm is enabled by default, `t_for_ack` is not zero. With a typical receive window of 4096, 50 is not greater than 4096. These 50 bytes remain in the send buffer, probably until another byte is received. This ACK will probably be delayed, causing more delay in sending the final 50 bytes.

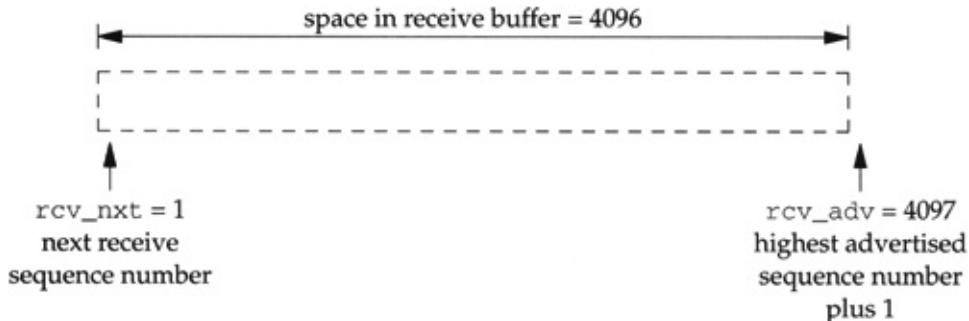
This example shows the timing delays that can occur when sending full-sized segments with the Nagle algorithm enabled. See [Figure 26.12](#).

## Example

This example demonstrates the ACK-every-other-byte behavior. Assume a connection is established with a segment size of 512 bytes and a receive buffer size of 4096. There is no data to receive.

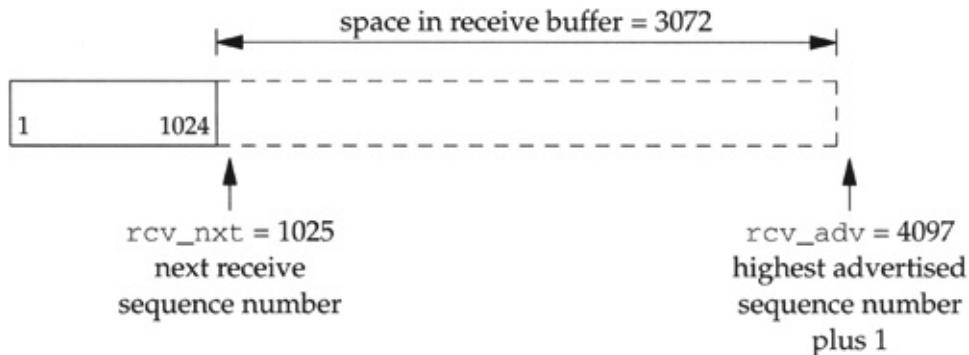
A window of 4096 is advertised in the ACK of the previous segment. The code in [Figure 26.8](#) shows the two variables `rcv_nxt` and `rcv_adv`. The value of

## Figure 26.13. Receiver advertising



The other end sends a segment with bytes 110 segment, sets the delayed-ACK flag for the con 1024 bytes of data to the socket's receiver buff updated as shown in [Figure 26.14](#).

## Figure 26.14. Transition from Figure 26.13



The process reads the 1024 bytes in its socket [Figure 30.6](#) that the resulting PRU\_RCVD request is called, because a window update might need to read data from the receive buffer. When tcp\_o

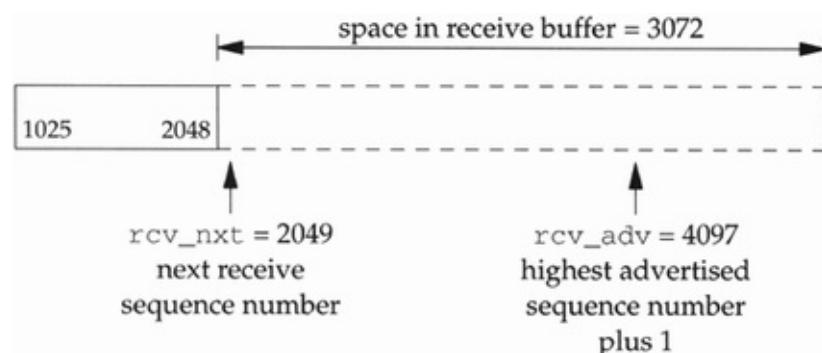
variables still have the values shown in Figure 2. The difference is that the amount of space in the receive buffer has increased by 1024 bytes since the process has read the first 1024 bytes. The calculation performed:

$$\begin{aligned} \text{adv} &= \min(4096, 65535) - (4097 - 1024) \\ &= 1024 \end{aligned}$$

TCP\_MAXWIN is 65535 and we assume a receiving window of 4096 bytes. Since the window has increased by less than two bytes, no ACK will be sent. But the delayed-ACK flag is still set, so if the ACK is delayed, it will be sent when the ACK will be sent.

When TCP receives the next segment with byte sequence number 2049, it processes the segment, sets the delayed-ACK flag (since the ACK was already on), and appends the 1024 bytes to the receive buffer. `rcv_nxt` is updated as shown in Figure 2.

**Figure 26.15. Transition from Figure 26.14 after reading 1024 bytes.**

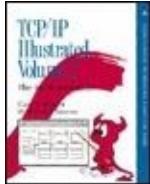


The process reads bytes 10252048 and `tcp_out` variables still have the values shown in [Figure 26.8](#): the receive buffer increases to 4096 when the process reads two segments of data. The calculations in [Figure 26.9](#) are performed as follows:

$$\begin{aligned} \text{adv} &= \min(4096, 65535) - (4096 - 2049) \\ &= 2048 \end{aligned}$$

This value is now greater than or equal to two segments, so the receiver sends an acknowledgment field of 2049 and the window is updated to 4096. This is a window update. The receiver is now acknowledged up to sequence number 6145. We'll see later in this function that the value of `rcv_adv` also gets updated to 6145.

This example shows that when receiving data faster than the ACK timer, an ACK is sent when the receive window has received two segments due to the process reading the data from the connection but the process is not reading the data from the receive buffer, the ACK-every-other-segment property means the sender will only see the delayed ACKs, each acknowledged until the receive buffer is filled and the window is updated.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

### 26.4 TCP Options

The TCP header can contain options. We digress to discuss these options since the next piece of `tcp_output` decides which options to send and constructs the options in the outgoing segment. [Figure 26.16](#) shows the format of the options supported by Net/3.

**Figure 26.16. TCP options supported by Net/3.**

End of option list:	<table border="1"><tr><td>kind=0</td></tr><tr><td>1 byte</td></tr></table>	kind=0	1 byte						
kind=0									
1 byte									
No operation:	<table border="1"><tr><td>kind=1</td></tr><tr><td>1 byte</td></tr></table>	kind=1	1 byte						
kind=1									
1 byte									
Maximum segment size:	<table border="1"><tr><td>kind=2</td><td>len=4</td><td>maximum segment size (MSS)</td></tr><tr><td>1 byte</td><td>1 byte</td><td>2 bytes</td></tr></table>	kind=2	len=4	maximum segment size (MSS)	1 byte	1 byte	2 bytes		
kind=2	len=4	maximum segment size (MSS)							
1 byte	1 byte	2 bytes							
Window scale factor:	<table border="1"><tr><td>kind=3</td><td>len=3</td><td>shift count</td></tr><tr><td>1 byte</td><td>1 byte</td><td>1 byte</td></tr></table>	kind=3	len=3	shift count	1 byte	1 byte	1 byte		
kind=3	len=3	shift count							
1 byte	1 byte	1 byte							
Timestamp:	<table border="1"><tr><td>kind=8</td><td>len=10</td><td>timestamp value</td><td>timestamp echo reply</td></tr><tr><td>1 byte</td><td>1 byte</td><td>4 bytes</td><td>4 bytes</td></tr></table>	kind=8	len=10	timestamp value	timestamp echo reply	1 byte	1 byte	4 bytes	4 bytes
kind=8	len=10	timestamp value	timestamp echo reply						
1 byte	1 byte	4 bytes	4 bytes						

Every option begins with a 1-byte *kind* that specifies the type of option. The first two options (with *kinds* of 0 and 1) are single-byte options. The other three are multibyte options with a *len* byte that follows the *kind* byte. The length is the total length, including the *kind* and *len* bytes.

The multibyte integers the MSS and the two timestamp values are stored in network byte order.

The final two options, window scale and timestamp, are new and therefore not supported by many systems. To provide interoperability with these older systems,

the following rules apply.

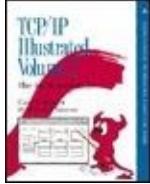
**1. TCP can send one of these options (or both) with the initial SYN segment corresponding to an active open (that is, a SYN without an ACK). Net/3 does this for both options if the global `tcp_do_rfc1323` is nonzero (it defaults to 1). This is done in `tcp_newtcpcb`.**

- The option is enabled only if the SYN reply from the other end also includes the desired option. This is handled in [Figures 28.20](#) and [29.2](#).
- If TCP performs a passive open and receives a SYN specifying the option, the response (the SYN plus ACK) must contain the option if TCP wants to enable the option. This is done in [Figure 26.23](#).

Since a system must ignore options that it doesn't understand, the newer options are enabled by both ends only if both ends understand the option and both ends want the option enabled.

The processing of the MSS option is covered in [Section 27.5](#). The next two sections summarize the Net/3 handling of the two newer options: window scale and timestamp.

Other options have been proposed. *kinds* of 4, 5, 6, and 7, called the selective-ACK and echo options, are defined in RFC 1072 [[Jacobson and Braden 1988](#)]. We don't show them in [Figure 26.16](#) because the echo options were replaced with the timestamp option, and selective ACKs, as currently defined, are still under discussion and were not included in RFC 1323. Also, the T/TCP proposal for TCP transactions (RFC 1644 [[Braden 1994](#)]), and Section 24.7 of Volume 1) specifies three options with *kinds* of 11, 12, and 13.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

### 26.5 Window Scale Option

The window scale option, defined in RFC 1323, avoids the limitation of a 16-bit window size field in the TCP header ([Figure 24.10](#)). Larger windows are required for what are called *long fat pipes*, networks with either a high bandwidth or a long delay (i.e., a long RTT). Section 24.3 of Volume 1 gives examples of current networks that require larger windows to obtain maximum TCP throughput.

The 1-byte shift count in [Figure 26.16](#) is between 0 (no scaling performed) and 14. This maximum value of 14 provides a maximum window of  $1,073,725,440$  bytes ( $65535 \times 2^{14}$ ). Internally Net/3 maintains

window sizes as 32-bit values, not 16-bit values.

The window scale option can only appear in a SYN segment; therefore the scale factor is fixed in each direction when the connection is established.

The two variables `snd_scale` and `rcv_scale` in the TCP control block specify the shift count for the send window and the receive window, respectively. Both default to 0 for no scaling. Every 16-bit advertised window received from the other end is left shifted by `snd_scale` bits to obtain the real 32-bit advertised window size ([Figure 28.6](#)).

Every time TCP sends a window advertisement to the other end, the internal 32-bit window size is right shifted by `rcv_scale` bits to give the value that is placed into the TCP header ([Figure 26.29](#)).

When TCP sends a SYN, either actively or passively, it chooses the value of `rcv_scale` to request, based on the size of the socket's receive buffer ([Figures 28.7](#) and [30.4](#)).

---

**Team-Fly** 

[◀ Previous](#)

[Next ▶](#)

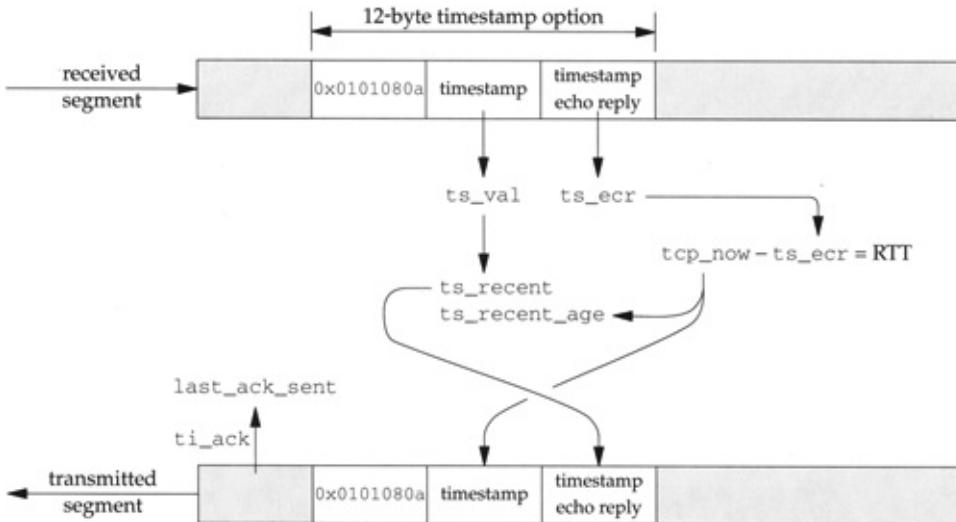
[Top](#)

## Chapter 26. TCP Output

### 26.6 Timestamp Option

The timestamp option is also defined in RFC 1323. In this option, the sender places a timestamp in every segment it sends, and the receiver sends the timestamp back in the acknowledgment, allowing the sender to calculate the round-trip time for each received ACK. [Figure 26.17](#) summarizes the variables used with the timestamp option and the variables involved.

**Figure 26.17. Summary of variables used with the timestamp option.**



The global variable  $tcp\_now$  is the timestamp copied from the received segment. It is initialized to 0 when the kernel is initialized and incremented by 1 every 500 ms (Figure 25.8). The timestamp options are maintained in the TCP control block.

- $ts\_recent$  is a copy of the most-recent valid timestamp from the other end. (We describe shortly what it means for a timestamp "valid.")
- $ts\_recent\_age$  is the value of  $tcp\_now$  when  $ts\_recent$  was last copied from a received segment.
- $last\_ack\_sent$  is the value of the acknowledgement number ( $ti\_ack$ ) the last time a segment was sent (Figure 26.32). This is normally equal to  $rcv\_nxt$ , the expected sequence number, unless ACKs are lost.

The two variables  $ts\_val$  and  $ts\_ecr$  are local variables.

the function `tcp_input` that contain the two values timestamp option.

- `ts_val` is the timestamp sent by the other end in the data.
- `ts_ecr` is the timestamp from the segment that was acknowledged by the received segment.

In an outgoing segment, the first 4 bytes of the timestamp option are set to 0x0101080a. This is the recommended value from [Appendix A](#) of RFC 1323. The 2 bytes of NOPs from [Figure 26.16](#), followed by a *kind* of 10, which identify the timestamp option. By inserting NOPs in front of the option, the two 32-bit time values in the option and the data that follows are aligned to byte boundaries. Also, we show the received timestamp in [Figure 26.17](#) with the recommended 12-byte format (which Net/3 always generates), but the code that processes received options ([Figure 28.10](#)) does not require this format. The 10-byte format shown in [Figure 26.16](#) without two preceding NOPs, is handled fine on Net/3 (see [Exercise 28.4](#)).

The RTT of a transmitted segment and its ACK is calculated as `tcp_now` minus `ts_ecr`. The units are 500-ms since that is the units of the Net/3 timestamps.

The presence of the timestamp option also allows to perform PAWS: protection against wrapped sequence numbers. We describe this algorithm in [Section](#). The variable `ts_recent_age` is used with PAWS.

`tcp_output` builds a timestamp option in an output segment by copying `tcp_now` into the timestamp and `ts_recent` into the echo reply ([Figure 26.24](#)). The value for every segment when the option is in use, unless the RST flag is set.

## Which Timestamp to Echo, RFC 1323 Algorithm

The test for a valid timestamp determines whether the value in `ts_recent` is updated, and since this value is always sent as the timestamp echo reply, the test's validity determines which timestamp gets echoed at the other end. RFC 1323 specified the following condition:

```
ti_seq <= last_ack_sent < ti_seq
```

which is implemented in C as shown in [Figure 26.18](#).

**Figure 26.18. Typical code to determine if a timestamp is valid.**

---

```

if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
    SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
    tp->ts_recent_age = tcp_now;
    tp->ts_recent = ts_val;
}

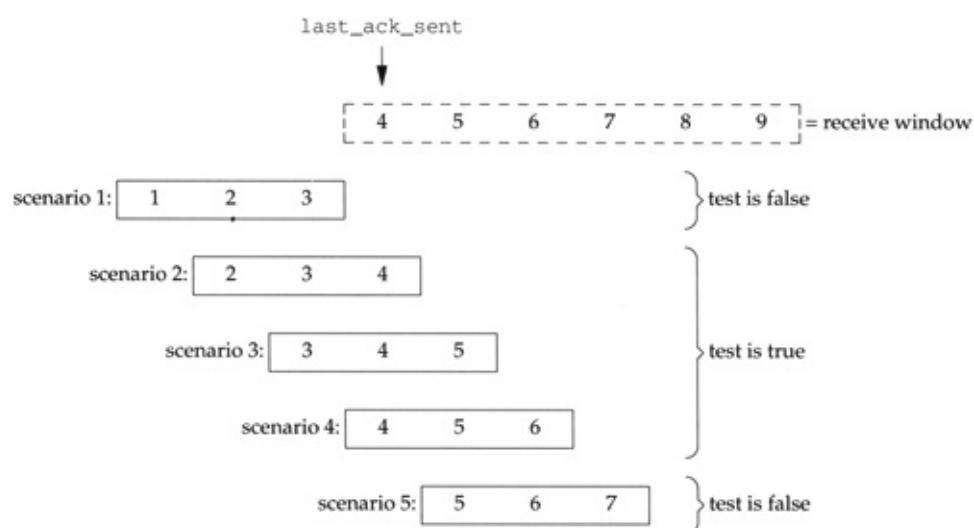
```

---

The variable `ts_present` is true if a timestamp is received in the segment. We encounter this code in `tcp_input`: [Figure 28.11](#) does the test in the header prediction code, and [Figure 28.35](#) does the test in normal input processing.

To see what this test is doing, [Figure 26.19](#) shows different scenarios, corresponding to five different segments received on a connection. In each scenario, `last_ack_sent` is 3.

**Figure 26.19. Example receive window and five scenarios of received segment.**



The left edge of the receive window begins with number 4. In scenario 1 the segment contains duplicate data. The SEQ\_LEQ test in [Figure 28](#), but the SEQ\_LT test fails. For scenarios 2, 3, and 4 the SEQ\_LEQ and SEQ\_LT tests are true because the left edge of the window is advanced by any one of the two segments, even though scenario 2 contains two bytes of data, and scenario 3 contains one duplicate byte of data. Scenario 5 fails the SEQ\_LEQ test, because it doesn't advance the left edge of the window. This is one in the future that's not the next expected sequence number, indicating that a previous segment was lost or reordered.

Unfortunately this test to determine whether to update ts\_recent is flawed [[Braden 1993](#)]. Consider the following example.

1. In [Figure 26.19](#) a segment that we don't expect arrives with bytes 1, 2, and 3. The times this segment is saved in ts\_recent because last\_ack\_sent is 1. An ACK is sent with an acknowledgment field of 4, and last\_ack set to 4 (the value of rcv\_nxt). We have the receive window shown in [Figure 26.19](#).

- This ACK is lost.
- The other end times out and retransmits the segment.

with bytes 1, 2, and 3. This segment arrives and is labeled "scenario 1" in [Figure 26.19](#). Since the test in [Figure 26.18](#) fails, `ts_recent` is not updated with the value from the retransmitted segment.

- A duplicate ACK is sent with an acknowledgement number 4, but the timestamp echo reply is `ts_recent`, which was copied from the segment in step 1. But when the receiver calculates the RTT using this value, it will (incorrectly) take into account the original transmission, the lost segment, the timeout, the retransmission, and the duplicate ACK.

For correct RTT estimation by the other end, the timestamp value from the retransmission should be returned in the duplicate ACK.

The tests in [Figure 26.18](#) also fail to update `ts_recent`. The length of the received segment is 0, since the window is not moved. This incorrect test can lead to problems with long-lived connections (greater than 24 days, the PAWS limit described in [Section 28.7](#)), unidirectional connections (all the data flow is in one direction, so the sender of the data always sends the same ACK).

## Which Timestamp to Echo, Corrected Algorithm

The algorithm we'll encounter in the Net/3 source code is:

[Figure 26.18.](#) The correct algorithm given in [B] replaces [Figure 26.18](#) with the one in [Figure 26](#)

## Figure 26.20. Correct code to determine if timestamp is valid.

```
if (ts_present && TSTMP_GEQ(ts_val, tp->ts_recent) &&  
SEQ_LEQ(ti->ti_seq, tp->last_ack_sent)) {
```

This doesn't test whether the left edge of the window moves or not, it just verifies that the new timestamp (ts\_val) is greater than or equal to the previous timestamp (ts\_recent), and that the starting sequence number of the received segment is not greater than the left edge of the window. Scenario 5 in [Figure 26.19](#) would fail to pass since it is out of order.

The macro TSTMP\_GEQ is identical to SEQ\_GEQ in [Figure 24.21](#). It is used with timestamps, since timestamps are 32-bit unsigned values that wrap around just like sequence numbers.

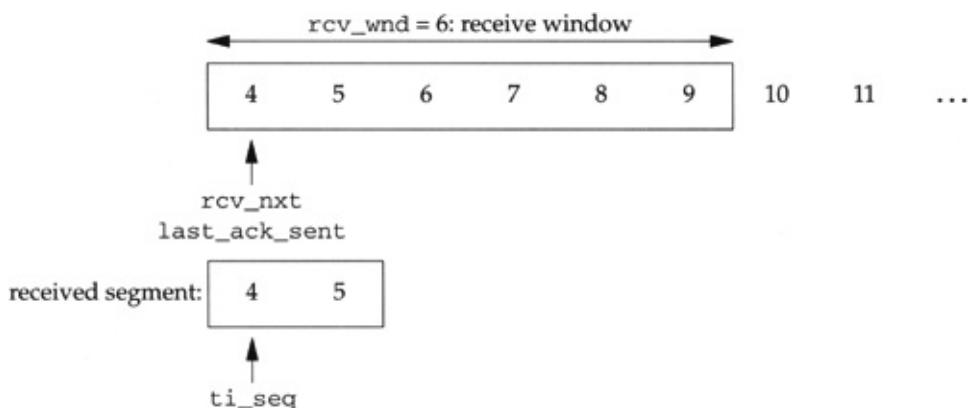
## Timestamps and Delayed ACKs

It is constructive to see how timestamps and R<sub>tt</sub> calculations are affected by delayed ACKs. Recall [Figure 26.17](#) that the value saved by TCP in ts\_

becomes the echoed timestamp in segments that are used by the other end in calculating i. When ACKs are delayed, the delay time should be taken into account by the side that sees the delays, or it might retransmit too quickly. In the example that we only consider the code in [Figure 26.20](#), but the incorrect code in [Figure 26.18](#) also handles delays correctly.

Consider the receive sequence space in [Figure 26.21](#). The received segment contains bytes 4 and 5.

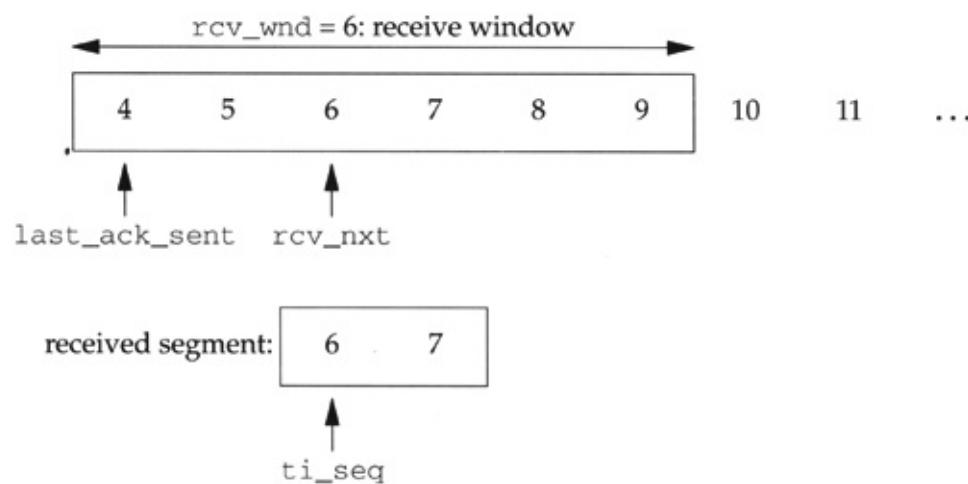
**Figure 26.21. Receive sequence space when a segment with bytes 4 and 5 arrives.**



Since `ti_seq` is less than or equal to `last_ack_seq`, `ts_recent` is copied from the segment. `recv_nxt` is increased by 2.

Assume that the ACK for these 2 bytes is delayed before that delayed ACK is sent, the next in-order arrives. This is shown in [Figure 26.22](#).

**Figure 26.22. Receive sequence space when with bytes 6 and 7 arrives.**



This time `ti_seq` is greater than `last_ack_sent`, it is not updated. This is intentional. Assuming TCP sends an ACK for sequence numbers 47, the other RTT will take into account the delayed ACK, since the echoed timestamp ([Figure 26.24](#)) is the one from the segment with sequence numbers 4 and 5. These also demonstrate that `recv_nxt` equals `last_ack_sent` when ACKs are delayed.

## Chapter 26. TCP Output

### 26.7 Send a Segment

The last half of `tcp_output` sends the segment with the TCP header and passes the segment to IP for delivery.

Figure 26.23 shows the first part, which sends options and scale options with a SYN segment.

**Figure 26.23. `tcp_output` function: send option and scale options with a SYN segment.**

---

```

222 send:
223 /*
224 * Before ESTABLISHED, force sending of initial options
225 * unless TCP set not to do any options.
226 * NOTE: we assume that the IP/TCP header plus TCP options
227 * always fit in a single mbuf, leaving room for a maximum
228 * link header, i.e.
229 * max_linkhdr + sizeof (struct tciphdr) + optlen <= MHLEN
230 */
231 optlen = 0;
232 hdrlen = sizeof(struct tciphdr);
233 if (flags & TH_SYN) {
234     tp->snd_nxt = tp->iss;
235     if ((tp->t_flags & TF_NOOPT) == 0) {
236         u_short mss;
237         opt[0] = TCPOPT_MAXSEG;
238         opt[1] = 4;
239         mss = htons((u_short) tcp_mss(tp, 0));
240         bcopy((caddr_t) & mss, (caddr_t) (opt + 2), sizeof(mss));
241         optlen = 4;
242         if ((tp->t_flags & TF_REQ_SCALE) &&
243             ((flags & TH_ACK) == 0 ||
244              (tp->t_flags & TF_RCVD_SCALE))) {
245             *((u_long *) (opt + optlen)) = htonl(TCPOPT_NOP << 24 |
246                                                 TCPOPT_WINDOW << 16 |
247                                                 TCPOLEN_WINDOW << 8 |
248                                                 tp->request_r_scale);
249             optlen += 4;
250         }
251     }
252 }

```

---

tcp\_output.c

## 223-234

The TCP options are built in the array opt, and keeps a count of the number of bytes accumulated (the options can be sent at once). If the SYN flag bit is set, it is added to the initial send sequence number (iss). If TCP is performing an active open, iss is set by the PRU\_CONNECT request. A TCP control block is created. If this is a passive open, it is added to the TCP control block and sets iss. In both cases, it is added to the global tcp\_iss.

## 235

The flag TF\_NOOPT is checked, but this flag is ignored.

there is no way to turn it on. Hence, the MSS option was always sent with a SYN segment.

In the Net/1 version of `tcp_newtcpcb`, the comment "TF\_NOOPT options!" appeared on the line that initialized the `TF_NOOPT` flag. This flag is probably a historical artifact from a system that had problems interoperating with other systems that did not send the MSS option, so the default was to not send it.

## Build MSS option

236-241

`opt[0]` is set to 2 (`TCPOPT_MAXSEG`) and `opt[1]` contains the length of the MSS option in bytes. The function `tcp_build_mss` builds the MSS to announce to the other end; we cover this in [Section 27.5](#). The 16-bit MSS is stored in `opt[2]` ([Exercise 26.5](#)). Notice that Net/3 always sends an announcement with the SYN for a connection.

## Should window scale option be sent?

242-244

If TCP is to request the window scale option, then this is an active open (`TH_ACK` is not set) or if the peer supports it and the window scale option was received in the SYN segment.

end. Recall that `t_flags` was set to `TF_REQ_SC` when the TCP control block was created in [Figure 26.23](#). The variable `tcp_do_rfc1323` was nonzero (its default value).

## Build window scale option

245-249

Since the window scale option occupies 3 bytes, a byte NOP is stored before the option, forcing the segment to be aligned on a 4-byte boundary. This causes the data in the segment to be aligned on a 4-byte boundary. If this is an active open, the window scale factor is calculated by the PRU\_CONNINFO function. If this is a passive open, the window scale factor is calculated when the SYN is received.

RFC 1323 specifies that if TCP is prepared to send this option even if its own shift count is 0. The option serves two purposes: to notify the other side of the option, and to announce its shift count. Even if one host calculates its own shift count as 0, the other encodes a different value.

The next part of `tcp_output` is shown in [Figure 26.24](#), which shows the code for building the options in the outgoing segment.

**Figure 26.24. `tcp_output` function: finish**

```

253  /*
254   * Send a timestamp and echo-reply if this is a SYN and our side
255   * wants to use timestamps (TF_REQ_TSTMP is set) or both our side
256   * and our peer have sent timestamps in our SYN's.
257   */
258  if ((tp->t_flags & (TF_REQ_TSTMP | TF_NOOPT)) == TF_REQ_TSTMP &&
259      (flags & TH_RST) == 0 &&
260      ((flags & (TH_SYN | TH_ACK)) == TH_SYN ||  

261       (tp->t_flags & TF_RCVD_TSTMP))) {
262      u_long *lp = (u_long *) (opt + optlen);
263
264      /* Form timestamp option as shown in appendix A of RFC 1323. */
265      *lp++ = htonl(TCPOPT_TSTAMP_HDR);
266      *lp++ = htonl(tcp_now);
267      *lp = htonl(tp->ts_recent);
268      optlen += TCPOLEN_TSTAMP_APPA;
269  }
270
271  /*
272   * Adjust data length if insertion of options will
273   * bump the packet length beyond the t_maxseg length.
274   */
275  if (len > tp->t_maxseg - optlen) {
276      len = tp->t_maxseg - optlen;
277      sendalot = 1;
278  }

```

tcp\_output.c

## Should timestamp option be sent?

253-261

If the following three conditions are all true, a timestamp option is sent: (1) TCP is configured to request the timestamp option (i.e., flags specifies the TF\_REQ\_TSTMP flag); (2) the segment being formed does not contain the RST flag; (3) either this is an active open (i.e., flags specifies the SYN flag) or TCP has received a timestamp from the peer (TF\_RCVD\_TSTMP). Unlike the MSS and window scale options, the timestamp option can be sent with every segment, even if the peers do not agree to use the option.

## Build timestamp option

263-267

The timestamp option ([Section 26.6](#)) consists of two bytes (TCPOLEN\_TSTAMP\_APPA). The first 4 bytes are constant TCPOPT\_TSTAMP\_HDR), as described. The timestamp value is taken from tcp\_now (the number of ticks since the system was initialized), and the scale factor is taken from ts\_recent, which is set by tcp\_inp.

## Check if options have overflowed segment

270-277

The size of the TCP header is incremented by the total bytes (optlen). If the amount of data to send (l) minus the size of the options (optlen), the data is reduced accordingly and the sendalot flag is set, to force this function after this segment is sent ([Figure 26.10](#)).

The MSS and window scale options only appear in segments which Net/3 always sends without data, so this data length doesn't apply. When the timestamp option however, it appears in all segments. This reduces the data length in each full-sized data segment from the announced MSS minus 12 bytes.

The next part of tcp\_output, shown in [Figure 26.11](#), performs statistics and allocates an mbuf for the IP and TCP headers.

is executed when the segment being output contains options (i.e., when `so->so_snd.sb_cc` is greater than 0).

## Figure 26.25. `tcp_output` function: update statistics for IP and TCP headers.

```
278  /*
279   * Grab a header mbuf, attaching a copy of data to
280   * be transmitted, and initialize the header from
281   * the template for sends on this connection.
282   */
283 if (len) {
284     if (tp->t_force && len == 1)
285         tcpstat.tcp_sndprobe++;
286     else if (SEQ_LT(tp->snd_nxt, tp->snd_max)) {
287         tcpstat.tcp_sndrexmitpack++;
288         tcpstat.tcp_sndrexmitbyte += len;
289     } else {
290         tcpstat.tcp_sndpack++;
291         tcpstat.tcp_sndbyte += len;
292     }
293     MGETHDR(m, M_DONTWAIT, MT_HEADER);
294     if (m == NULL) {
295         error = ENOBUFS;
296         goto out;
297     }
298     m->m_data += max_linkhdr;
299     m->m_len = hdrlen;
300     if (len <= MHLEN - hdrlen - max_linkhdr) {
301         m_copydata(so->so_snd.sb_mb, off, (int) len,
302                     mtod(m, caddr_t) + hdrlen);
303         m->m_len += len;
304     } else {
305         m->m_next = m_copy(so->so_snd.sb_mb, off, (int) len);
306         if (m->m_next == 0)
307             len = 0;
308     }
309     /*
310      * If we're sending everything we've got, set PUSH.
311      * (This will keep happy those implementations that
312      * give data to the user only when a buffer fills or
313      * a PUSH comes in.)
314      */
315     if (off + len == so->so_snd.sb_cc)
316         flags |= TH_PUSH;
```

## Update statistics

284-292

If `t_force` is nonzero and TCP is sending a single window probe. If `snd_nxt` is less than `snd_max` retransmission. Otherwise, this is normal data transmission.

## Allocate an mbuf for IP and TCP headers

293-297

An mbuf with a packet header is allocated by `M_ALLOC` for the IP and TCP headers, and possibly the data payload. Although `tcp_output` is often called as part of a write operation (e.g., when a user-space application performs a write) it is also called at the software interrupt level as part of the timer processing. Therefore `M_DONTWAIT` is used. If an error is returned, a jump is made to the label `l_error` near the end of the function, in [Figure 26.32](#).

## Copy data into mbuf

298-308

If the amount of data is less than 44 bytes (10 bytes for TCP options), the data is copied directly from the source mbuf into the new packet header mbuf by `m_copydata`. If the data is more than 44 bytes, `m_copydata` creates a new mbuf chain with the data from the source mbuf and this chain is linked to the new packet header mbuf. For a detailed description of `m_copy` in [Section 2.9](#), where we saw that if the data is in a cluster, `m_copy` just references that cluster.

make a copy of the data.

## Set PSH flag

309-316

If TCP is sending everything it has from the server, the PSH flag is set. As the comment indicates, this is intended for systems that only pass received data to an application when a flag is received or when a buffer fills. We'll see that TCP never holds data in a socket receive buffer waiting for a PSH flag.

The next part of `tcp_output`, shown in [Figure 26.26](#), is the code that is executed when `len` equals 0: there is no segment and TCP is sending.

**Figure 26.26. `tcp_output` function: update structure of mbuf for IP and TCP headers**

```

317     } else { /* len == 0 */
318         if (tp->t_flags & TF_ACKNOW)
319             tcpstat.tcpstat_sndacks++;
320         else if (flags & (TH_SYN | TH_FIN | TH_RST))
321             tcpstat.tcpstat_sndctrl++;
322         else if (SEQ_GT(tp->snd_up, tp->snd_una))
323             tcpstat.tcpstat_sndurg++;
324         else
325             tcpstat.tcpstat_sndwinup++;
326
327         MGETHDR(m, M_DONTWAIT, MT_HEADER);
328         if (m == NULL) {
329             error = ENOBUFS;
330             goto out;
331         }
332         m->m_data += max_linkhdr;
333         m->m_len = hdrlen;
334     }
335     m->m_pkthdr.rcvif = (struct ifnet *) 0;
336     ti = mtod(m, struct tciphdr *);
337     if (tp->t_template == 0)
338         panic("tcp_output");
339     bcopy((caddr_t) tp->t_template, (caddr_t) ti, sizeof(struct tciphdr));

```

*tcp\_output.c*

## Update statistics

318-325

Various statistics are updated: TF\_ACKNOW and this is an ACK-only segment. If any one of the flags is set, this is a control segment. If the urgent pointer is greater than the snd\_una, the segment is being sent to notify the receiver of an urgent pointer. If none of these conditions are true, it's a window update.

## Get mbuf for IP and TCP headers

326-335

An mbuf with a packet header is allocated to contain

headers.

## **Copy IP and TCP header templates into mbuf**

336-338

The template of the IP and TCP headers is copied into the mbuf by bcopy. This template was created

[Figure 26.27](#) shows the next part of `tcp_output` setting the remaining fields in the TCP header.

**Figure 26.27. `tcp_output` function: set `ti_seq`**

```

339  /*
340   * Fill in fields, remembering maximum advertised
341   * window for use in delaying messages about window sizes.
342   * If resending a FIN, be sure not to use a new sequence number.
343   */
344  if (flags & TH_FIN && tp->t_flags & TF_SENTPIN &
345      tp->snd_nxt == tp->snd_max)
346      tp->snd_nxt--;
347  /*
348   * If we are doing retransmissions, then snd_nxt will
349   * not reflect the first unsent octet. For ACK only
350   * packets, we do not want the sequence number of the
351   * retransmitted packet, we want the sequence number
352   * of the next unsent octet. So, if there is no data
353   * (and no SYN or FIN), use snd_max instead of snd_nxt
354   * when filling in ti_seq. But if we are in persist
355   * state, snd_max might reflect one byte beyond the
356   * right edge of the window, so use snd_nxt in that
357   * case, since we know we aren't doing a retransmission.
358   * (retransmit and persist are mutually exclusive...)
359   */
360  if (len || (flags & (TH_SYN | TH_FIN)) || tp->t_timer[TCPT_PERSIST])
361      ti->ti_seq = htonl(tp->snd_nxt);
362  else
363      ti->ti_seq = htonl(tp->snd_max);
364  ti->ti_ack = htonl(tp->recv_nxt);
365  if (optlen) {
366      bcopy((caddr_t) opt, (caddr_t) (ti + 1), optlen);
367      ti->ti_off = (sizeof(struct tcphdr) + optlen) >> 2;
368  }
369  ti->ti_flags = flags;

```

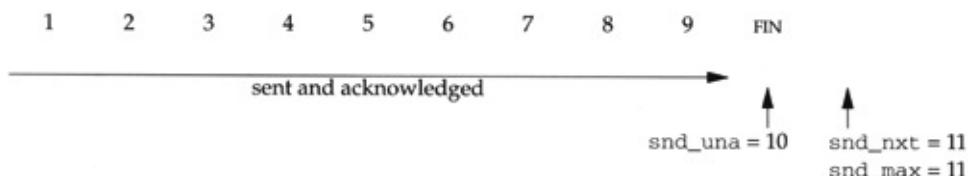
tcp\_output.c

## Decrement snd\_nxt if FIN is being retransmit

339-346

If TCP has already transmitted the FIN, the sen appears as shown in [Figure 26.28](#).

## Figure 26.28. Send sequence space after FIN



Therefore, if the FIN flag is set, and if the TF\_S and if snd\_nxt equals snd\_max, TCP knows the retransmitted. We'll see shortly ([Figure 26.31](#)) sent, snd\_nxt is incremented 1 one (since the F sequence number), so this piece of code decrements it by 1.

## **Set sequence number field of segment**

347-363

The sequence number field of the segment is normally set to snd\_nxt, but is set to snd\_max if (1) there is no outstanding segment (i.e., n equals 0), (2) neither the SYN flag nor the FIN flag is set, and (3) the persist timer is not set.

## **Set acknowledgment field of segment**

364

The acknowledgment field of the segment is always set to the next expected receive sequence number.

## **Set header length if options present**

365-368

If TCP options are present (optlen is greater than zero), the options are copied into the TCP header and the 4-bit header length field (th\_off in [Figure 24.10](#)) is set to the fixed header length (20 bytes) plus the length of the option field (which is the number of 32-bit words in the TCP header times optlen).

369

The flags field in the TCP header is set from the value of the flags parameter.

The next part of code, shown in [Figure 26.29](#), fills in the TCP header and calculates the TCP checksum.

**Figure 26.29. `tcp_output` function: fill in more fields and calculate checksum**

```

370  /*
371   * Calculate receive window.  Don't shrink window,
372   * but avoid silly window syndrome.
373   */
374  if (win < (long) (so->so_rcv.sb_hiwat / 4) && win < (long) tp->t_maxseg)
375      win = 0;
376  if (win > (long) TCP_MAXWIN << tp->rcv_scale)
377      win = (long) TCP_MAXWIN << tp->rcv_scale;
378  if (win < (long) (tp->rcv_adv - tp->rcv_nxt))
379      win = (long) (tp->rcv_adv - tp->rcv_nxt);
380  ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
381  if (SEQ_GT(tp->snd_up, tp->snd_nxt)) {
382      ti->ti_urp = htons((u_short) (tp->snd_up - tp->snd_nxt));
383      ti->ti_flags |= TH_URG;
384  } else
385      /*
386       * If no urgent pointer to send, then we pull
387       * the urgent pointer to the left edge of the send window
388       * so that it doesn't drift into the send window on sequence
389       * number wraparound.
390       */
391  tp->snd_up = tp->snd_una; /* drag it along */
392  /*
393   * Put TCP length in extended header, and then
394   * checksum extended header and data.
395   */
396  if (len + optlen)
397      ti->ti_len = htons((u_short) (sizeof(struct tcphdr) +
398                               optlen + len));
399  ti->ti_sum = in_cksum(m, (int) (hdrlen + len));

```

tcp\_output.c

## Don't advertise less than one full-sized segm

370-375

Avoidance of the silly window syndrome is performed by calculating the window size that is advertised to the receiver (ti\_win). Recall that win was set at the end of  $\frac{1}{4}$  of the amount of space in the socket's receive buffer. If the receiver advertises a window that is less than a fourth of the receive buffer size (so\_rcv.sb\_hiwat), the advertised window will be shrunk to one full-sized segment, the advertised window will be shrunk to one full-sized segment. The later test that prevents the window from shrinking to one full-sized segment is the one that checks if the advertised window is less than one full-sized segment.

space will be advertised.

## Observe upper limit for advertised window or

376-377

If win is larger than the maximum value for this receiver, set it to its maximum value.

## Do not shrink window

378-379

Recall from [Figure 26.10](#) that  $rcv\_adv$  minus  $rcv\_seq$  is the amount of space still available to the sender that was previously advertised. If  $win$  is less than this value,  $win$  is set to this value and the receiver does not shrink the window. This can happen when the receiver has received less than one full-sized segment (hence  $win$  was not shrunk at the beginning of this figure), but there is room in the receiver's buffer for some data. Figure 22.3 of Volume 1 shows an example of this scenario.

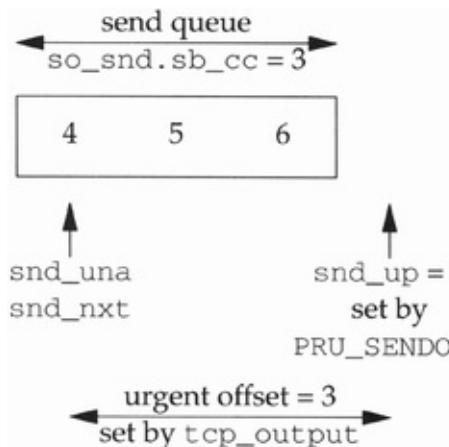
## Set urgent offset

381-383

If the urgent pointer (snd\_up) is greater than s urgent mode. The urgent offset in the TCP head offset of the urgent pointer from the starting se segment, and the URG flag bit is set. TCP sends the URG flag regardless of whether the referenced data is contained in this segment or not.

Figure 26.30 shows an example of how the urg calculated, assuming the process executes

### Figure 26.30. Example of urgent pointer calculation.



```
send(fd, buf, 3, MSG_OOB);
```

and the send buffer is empty when this call to s shows that Berkeley-derived systems consider point to the first byte of data *after* the out-of-b

discussion after [Figure 24.10](#) where we distinguish between the 32-bit *urgent pointer* in the data stream (`snd_una`) and the 16-bit *urgent offset* in the TCP header (`ti_urp`).

There is a subtle bug here. The bug occurs when the value of `ti_urp` is larger than 65535, regardless of whether the send buffer is in use or not. If the send buffer is greater than 65535 and the process sends out-of-band data, the urgent pointer from `snd_nxt` can exceed 65535. Since the urgent pointer is a 16-bit unsigned value, and if the value of `ti_urp` exceeds 65535, the 16 high-order bits are discarded, resulting in a bogus urgent pointer to the other end. See [Exercise 24.10](#) for a solution.

### 384-391

If TCP is not in urgent mode, the urgent pointer is set to the edge of the window (`snd_una`).

### 392-399

The TCP length is stored in the pseudo-header, and the checksum is calculated. All the fields in the TCP header have their initial values when the IP and TCP header template were copied into the socket header ([Figure 26.26](#)), the fields in the IP header that were not part of the pseudo-header were initialized (as shown in [Figure 26.27](#)), and the checksum calculation was performed.

The next part of `tcp_output`, shown in [Figure 26.28](#),

sequence number if the SYN or FIN flags are set, initialize retransmission timer.

## Figure 26.31. `tcp_output` function: update initialize retransmit time

```
400  /*
401   * In transmit state, time the transmission and arrange for
402   * the retransmit. In persist state, just set snd_max.
403   */
404  if (tp->t_force == 0 || tp->t_timer[TCPT_PERSIST] == 0) {
405      tcp_seq startseq = tp->snd_nxt;
406
407      /*
408       * Advance snd_nxt over sequence space of this segment.
409       */
410      if (flags & (TH_SYN | TH_FIN)) {
411          if (flags & TH_SYN)
412              tp->snd_nxt++;
413          if (flags & TH_FIN)
414              tp->snd_nxt++;
415          tp->t_flags |= TF_SENDFIN;
416      }
417      tp->snd_nxt += len;
418      if (SEQ_GT(tp->snd_nxt, tp->snd_max)) {
419          tp->snd_max = tp->snd_nxt;
420
421          /*
422           * Time this transmission if not a retransmission and
423           * not currently timing anything.
424           */
425          if (tp->t_rtt == 0) {
426              tp->t_rtt = 1;
427              tp->t_rtseq = startseq;
428              tcpstat.tcpst_segstimed++;
429          }
430
431          /*
432           * Set retransmit timer if not currently set,
433           * and not doing an ack or a keepalive probe.
434           * Initial value for retransmit timer is smoothed
435           * round-trip time + 2 * round-trip time variance.
436           * Initialize counter which is used for backoff
437           * of retransmit time.
438           */
439          if (tp->t_timer[TCPT_REXMT] == 0 &&
440              tp->snd_nxt != tp->snd_una) {
441              tp->t_timer[TCPT_REXMT] = tp->t_rxtdelay;
442              if (tp->t_timer[TCPT_PERSIST]) {
443                  tp->t_timer[TCPT_PERSIST] = 0;
444                  tp->t_rxtshift = 0;
445              }
446          }
447      } else if (SEQ_GT(tp->snd_nxt + len, tp->snd_max))
448          tp->snd_max = tp->snd_nxt + len;

```

## **Remember starting sequence number**

**400-405**

If TCP is not in the persist state, the starting sequence number is saved in startseq. This is used later in Figure 2(e) to determine if a segment is timed.

## **Increment snd\_nxt**

**406-417**

Since both the SYN and FIN flags take a sequence number, snd\_nxt is incremented if either is set. TCP also remembers the sequence number of the last segment sent, by setting the flag TF\_SENTFIN. snd\_nxt is then incremented by the number of bytes of data (length).

## **Update snd\_max**

**418-419**

If the new value of snd\_nxt is larger than snd\_max, it becomes the new maximum. If the segment is not currently being timed for retransmission. The new value of snd\_max is stored in the variable.

**420-428**

If a segment is not currently being timed for retransmission, it is added to the list of segments being timed.

equals 0), the timer is started (`t_rtt` is set to 1) sequence number of the segment being timed is This sequence number is used by `tcp_input` to determine if the segment being timed is acknowledged, to update the sequence number. The sample code we discussed in [Section 25.1C](#)

```
if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->seq))  
    tcp_xmit_timer(tp, tp->t_rtt);
```

## Set retransmission timer

### 430-440

If the retransmission timer is not currently set, and the segment contains data, the retransmission timer is set to `t_rxtcur`. `t_rxtcur` is set by `tcp_xmit_timer`, when an RTT value is made. This is an ACK-only segment if `snd_nxt` (the sequence number of the segment whose length was added to `snd_nxt` earlier in this figure), the retransmission timer is set only for segments containing data.

### 441-444

If the persist timer is enabled, it is disabled. Either the retransmission timer or the persist timer can be enabled at any time during a connection, but not both.

## Persist state

**446-447**

The connection is in the persist state since t\_fo persist timer is enabled. (This else clause is at the beginning of the figure.) snd\_max is updated persist state, len will be one.

The final part of tcp\_output, shown in [Figure 26](#), forms the formation of the outgoing segment and calls ip\_datagram.

## Add trace record for socket debugging

**448-452**

If the SO\_DEBUG socket option is enabled, tcp\_ uses TCP's circular trace buffer. We describe this function in detail in [Section 2.10](#).

## Set IP length, TTL, and TOS

**453-462**

The final three fields in the IP header that must be set by the transport layer are stored: IP length, TTL, and TOS. The fields are marked with an asterisk at the bottom of the code.

The comments XXX are because the latter two fields are optional.

remain constant for a connection and should I header template, instead of being assigned e> segment is sent. But these two fields cannot t header until after the TCP checksum is calcula

## Pass datagram to IP

463-464

ip\_output sends the datagram containing the T socket options are logically ANDed with SO\_DON means that the only socket option passed to ip\_ SO\_DONTROUTE. The only other socket option ip\_output is SO\_BROADCAST, so this logical AN SO\_BROADCAST bit, if set. This means that a p connect to a broadcast address, even if it sets t socket option.

467-470

The error ENOBUFS is returned if the interface needs to obtain an mbuf and can't. The function connection into slow start, by setting the conge full-sized segment. Notice that tcp\_output still case, instead of the error, even though the data. This differs from udp\_output ([Figure 23.20](#)), wl The difference is that UDP is unreliable, so the

is the only indication to the process that the data has been transmitted. TCP, however, will time out (if the segment contains no data) and will retransmit the datagram, and it is hoped that the interface will queue the segment. If the interface output queue or more available memory becomes full before the segment is transmitted, the other end will receive an ACK isn't received and will retransmit the data, which will be discarded.

**Figure 26.32. `tcp_output` function: call `ip_out`**

```

448  /*
449   * Trace.
450   */
451   if (so->so_options & SO_DEBUG)
452     tcp_trace(TA_OUTPUT, tp->t_state, tp, ti, 0);
453
454  /*
455   * Fill in IP length and desired time to live and
456   * send to IP level. There should be a better way
457   * to handle ttl and tos; we could keep them in
458   * the template, but need a way to checksum without them.
459   */
460   m->m_pkthdr.len = hdrlen + len;
461   ((struct ip *) ti)->ip_len = m->m_pkthdr.len;
462   ((struct ip *) ti)->ip_ttl = tp->t_inpcb->inp_ip.ip_ttl; /* XXX */
463   ((struct ip *) ti)->ip_tos = tp->t_inpcb->inp_ip.ip_tos; /* XXX */
464   error = ip_output(m, tp->t_inpcb->inp_options, &tp->t_inpcb->inp_route,
465                     so->so_options & SO_DONTROUTE, 0);
466   if (error) {
467     out:
468     if (error == ENOBUFS) {
469       tcp_quench(tp->t_inpcb, 0);
470       return (0);
471     }
472     if ((error == EHOSTUNREACH || error == ENETDOWN)
473         && TCPS_HAVERCVDSYN(tp->t_state)) {
474       tp->t_software_error = error;
475       return (0);
476     }
477   }
478   tcpstat.tcp_sndtotal++;
479
480  /*
481   * Data sent (as far as we can tell).
482   * If this advertises a larger window than any other segment,
483   * then remember the size of the advertised window.
484   * Any pending ACK has now been sent.
485   */
486   if (win > 0 && SEQ_GT(tp->recv_nxt + win, tp->recv_adv))
487     tp->recv_adv = tp->recv_nxt + win;
488   tp->last_ack_sent = tp->recv_nxt;
489   tp->t_flags &= ~(TF_ACKNOW | TF_DELACK);
490
491   if (sendalot)
492     goto again;
493   return (0);
494 }
```

tcp\_output.c

## 471-475

If a route can't be located for the destination, and the connection has received a SYN, the error is recorded as a soft connection.

When `tcp_output` is called by `tcp_usrreq` as part of the PRU process ([Chapter 30](#), the PRU\_CONNECT, PRU\_

PRU\_SENDOOB, and PRU\_SHUTDOWN requests receives the return value from `tcp_output`. Other `tcp_output`, such as `tcp_input` and the fast and slow functions, ignore the return value (because they return an error to a process).

## **Update `rcv_adv` and `last_ack_sent`**

479-486

If the highest sequence number advertised in the segment plus `win`) is larger than `rcv_adv`, the new value of `rcv_adv` was used in [Figure 26.9](#) to determine how much window had opened since the last segment that was seen. This is done to make certain TCP was not shrinking the window.

487

The value of the acknowledgment field in the segment is stored in `last_ack_sent`. This variable is used by `tcp_input` to implement the `ACK` option ([Section 26.6](#)).

488

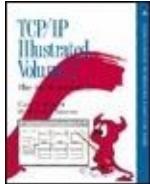
Any pending ACK has been sent, so the `TF_ACKED` flags are cleared.

## More data to send?

489-490

If the sendalot flag is set, a jump is made back ([Figure 26.1](#)). This occurs if the send buffer contains a full-sized segment that can be sent ([Figure 26.2](#)). A segment was being sent and TCP options were set to indicate the amount of data in the segment ([Figure 26.3](#)).

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

### 26.8 tcp\_template Function

The function `tcp_newtcpcb` (from the previous chapter) is called when the socket is created, to allocate and partially initialize the TCP control block. When the first segment is sent or received on the socket (an active open is performed, the `PRU_CONNECT` request, or a `SYN` arrives for a listening socket), `tcp_template` creates a template of the IP and TCP headers for the connection. This minimizes the amount of work required by `tcp_output` when a segment is sent on the connection.

[Figure 26.33](#) shows the `tcp_template` function.

## Figure 26.33. `tcp_template` function: create template of IP and TCP headers.

```
59 struct tciphdr *
60 tcp_template(tp)
61 struct tcpcb *tp;
62 {
63     struct inpcb *inp = tp->t_inpcb;
64     struct mbuf *m;
65     struct tciphdr *n;
66
67     if ((n = tp->t_template) == 0) {
68         m = m_get(M_DONTWAIT, MT_HEADER);
69         if (m == NULL)
70             return (0);
71         m->m_len = sizeof(struct tciphdr);
72         n = mtod(m, struct tciphdr *);
73     }
74     n->ti_next = n->ti_prev = 0;
75     n->ti_x1 = 0;
76     n->ti_pr = IPPROTO_TCP;
77     n->ti_len = htons(sizeof(struct tciphdr) - sizeof(struct ip));
78     n->ti_src = inp->inp_laddr;
79     n->ti_dst = inp->inp_faddr;
80     n->ti_sport = inp->inp_lport;
81     n->ti_dport = inp->inp_fport;
82     n->ti_seq = 0;
83     n->ti_ack = 0;
84     n->ti_x2 = 0;
85     n->ti_off = 5;           /* 5 32-bit words = 20 bytes */
86     n->ti_flags = 0;
87     n->ti_win = 0;
88     n->ti_sum = 0;
89     n->ti_urp = 0;
90     return (n);
91 }
```

## Allocate mbuf

59-72

The template of the IP and TCP headers is formed in an mbuf, and a pointer to the mbuf is stored in the `t_template` member of the TCP control block. Since this function can be called at the software

interrupt level, from `tcp_input`, the `M_DONTWAIT` flag is specified.

## Initialize header fields

73-88

All the fields in the IP and TCP headers are set to 0 except as follows: `ti_pr` is set to the IP protocol value for TCP (6); `ti_len` is set to 20, the default length of the TCP header; and `ti_off` is set to 5, the number of 32-bit words in the 20-byte TCP header. Also the source and destination IP addresses and TCP port numbers are copied from the Internet PCB into the TCP header template.

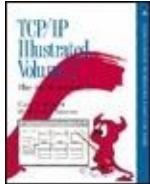
## Pseudo-header for TCP checksum computation

73-88

The initialization of many of the fields in the combined IP and TCP header simplifies the computation of the TCP checksum, using the same pseudo-header technique

as discussed for UDP in [Section 23.6](#). Examining the `udphdr` structure in [Figure 23.19](#) shows why `tcp_template` initializes fields such as `ti_next` and `ti_prev` to 0.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

### 26.9 tcp\_respond Function

The function `tcp_respond` is a special-purpose function that also calls `ip_output` to send IP datagrams. `tcp_respond` is called in two cases:

- 1. by `tcp_input` to generate an RST segment, with or without an ACK, and**

- by `tcp_timers` to send a keepalive probe.

Instead of going through all the logic of `tcp_output` for these two cases, the special-purpose function `tcp_respond` is called. We also note that the function `tcp_drop` that we cover in the next chapter also generates RST segments by calling

tcp\_output. Not all RST segments are generated by tcp\_respond.

Figure 26.34 shows the first half of tcp\_respond.

## Figure 26.34. tcp\_respond function: first half.

```
104 void  
105 tcp_respond(tp, ti, m, ack, seq, flags) tcp_subr.c  
106 struct tcpcb *tp;  
107 struct tcphdr *ti;  
108 struct mbuf *m;  
109 tcp_seq ack, seq;  
110 int flags;  
111 {  
112     int tlen;  
113     int win = 0;  
114     struct route *ro = 0;  
115     if (tp) {  
116         win = sbspace(&tp->t_inpcb->inp_socket->so_rcv);  
117         ro = &tp->t_inpcb->inp_route;  
118     }  
119     if (m == 0) { /* generate keepalive probe */  
120         m = m_gethdr(M_DONTWAIT, MT_HEADER);  
121         if (m == NULL)  
122             return;  
123         tlen = 0; /* no data is sent */  
124         m->m_data += max_linkhdr;  
125         *mtod(m, struct tcphdr *) = *ti;  
126         ti = mtod(m, struct tcphdr *);  
127         flags = TH_ACK;  
128     } else { /* generate RST segment */  
129         m_freem(m->m_next);  
130         m->m_next = 0;  
131         m->m_data = (caddr_t) ti;  
132         m->m_len = sizeof(struct tcphdr);  
133         tlen = 0;  
134 #define xchg(a,b,type) { type t; t=a; a=b; b=t; }  
135         xchg(ti->ti_dst.s_addr, ti->ti_src.s_addr, u_long);  
136         xchg(ti->ti_dport, ti->ti_sport, u_short);  
137 #undef xchg  
138 }
```

Figure 26.35 shows the different arguments to `tcp_respond` for the three cases in which it is called.

## Figure 26.35. Arguments to `tcp_respond`.

	Arguments					
	tp	ti	m	ack	seq	flags
generate RST without ACK	tp	ti	m	0	ti_ack	TH_RST
generate RST with ACK	tp	ti	m	ti_seq + ti_len	0	TH_RST   TH_ACK
generate keepalive	tp	t_template	NULL	recv_nxt	snd_una	0

`tp` is a pointer to the TCP control block (possibly a null pointer); `ti` is a pointer to an IP/TCP header template; `m` is a pointer to the mbuf containing the segment causing the RST to be generated; and the last three arguments are the acknowledgment field, sequence number field, and flags field of the segment being generated.

113-118

It is possible for `tcp_input` to generate an RST when a segment is received that does not have an associated TCP control block. This happens, for example, when a segment is received that doesn't reference

an existing connection (e.g., a SYN for a port without an associated listening server). In this case tp is null and the initial values for win and ro are used. If tp is not null, the amount of space in the receive buffer will be sent as the advertised window, and the pointer to the cached route is saved in ro for the call to ip\_output.

## **Send keepalive probe when keepalive timer expires**

119-127

The argument m is a pointer to the mbuf chain for the received segment. But a keep-alive probe is sent in response to the keepalive timer expiring, not in response to a received TCP segment. Therefore m is null and m\_gethdr allocates a packet header mbuf to contain the IP and TCP headers. tlen, the length of the TCP data, is set to 0, since the keepalive probe doesn't contain any data.

Some older implementations based on

4.2BSD do not respond to these keepalive probes unless the segment contains data. Net/3 can be configured to send 1 garbage byte of data in the probe to elicit the response by defining the name TCP\_COMPAT\_42 when the kernel is compiled. This assigns 1, instead of 0, to tlen. The garbage byte causes no harm, because it is not the expected byte (it is a byte that the receiver has previously received and acknowledged), so it is thrown away by the receiver.

The assignment of \*ti copies the TCP header template structure pointed to by ti into the data portion of the mbuf. The pointer ti is then set to point to the header template in the mbuf.

## **Send RST segment in response to received segment**

128-138

An RST segment is being sent by tcp\_input in response to a received segment. The

mbuf containing the input segment is reused for the response. All the mbufs on the chain are released by `m_free` except the first mbuf (the packet header), since the segment generated by `tcp_respond` consists of only an IP header and a TCP header. The source and destination IP address and port numbers are swapped in the IP and TCP header.

[Figure 26.36](#) shows the final half of `tcp_respond`.

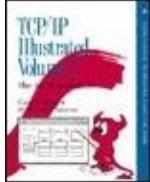
## Figure 26.36. `tcp_respond` function: second half.

```
139     ti->ti_len = htons((u_short) (sizeof(struct tcphdr) + tlen));          tcp_subr.c
140     tlen += sizeof(struct tcphdr);
141     m->m_len = tlen;
142     m->m_pkthdr.len = tlen;
143     m->m_pkthdr.rcvif = (struct ifnet *) 0;
144     ti->ti_next = ti->ti_prev = 0;
145     ti->ti_x1 = 0;
146     ti->ti_seq = htonl(seq);
147     ti->ti_ack = htonl(ack);
148     ti->ti_x2 = 0;
149     ti->ti_off = sizeof(struct tcphdr) >> 2;
150     ti->ti_flags = flags;
151     if (tp)
152         ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
153     else
154         ti->ti_win = htons((u_short) win);
155     ti->ti_urp = 0;
156     ti->ti_sum = 0;
157     ti->ti_sum = in_cksum(m, tlen);
158     ((struct ip *) ti)->ip_len = tlen;
159     ((struct ip *) ti)->ip_ttl = ip_defttl;
160     (void) ip_output(m, NULL, ro, 0, NULL);
161 }
```

The fields in the IP and TCP headers must be initialized for the TCP checksum computation. These statements are similar to the way `tcp_template` initializes the `t_template` field. The sequence number and acknowledgment fields are passed by the caller as arguments. Finally `ip_output` sends the datagram.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 26. TCP Output

### 26.10 Summary

This chapter has looked at the general-purpose function that generates most TCP segments (`tcp_output`) and the special-purpose function that generates RST segments and keepalive probes (`tcp_respond`).

Many factors determine whether TCP can send a segment or not: the flags in the segment, the window advertised by the other end, the amount of data ready to send, whether unacknowledged data already exists for the connection, and so on. Therefore the logic of `tcp_output` determines whether a segment can be sent (the first half of the function), and if

so, what values to set all the TCP header fields to (the last half of the function). If a segment is sent, the TCP control block variables for the send sequence space must be updated.

One segment at a time is generated by `tcp_output`, and at the end of the function a check is made of whether more data can still be sent. If so, the function loops around and tries to send another segment. This looping continues until there is no more data to send, or until some other condition (e.g., the receiver's advertised window) stops the transmission.

A TCP segment can also contain options. The options supported by Net/3 specify the maximum segment size, a window scale factor, and a pair of timestamps. The first two can only appear with SYN segments, while the timestamp option (if supported by both ends) normally appears in every segment. Since the window scale and timestamp options are newer and optional, if the first end to send a SYN wants to use the option, it sends the option with its SYN and uses the option only if the other end's

SYN also contains the option.

## Exercises

Slow start is resumed in [Figure 26.1](#) when there is a pause in the *sending* of data, yet the amount of idle time is calculated as the amount of time since the last **26.1** segment was *received* on the connection. Why doesn't TCP calculate the idle time as the amount of time since the last segment was *sent* on the connection?

With [Figure 26.6](#) we said that len is less than 0 if the FIN has been sent **26.2** but not acknowledged and not retransmitted. What happens if the FIN is retransmitted?

Net/3 always sends the window scale and timestamp options with

**26.3** an active open. Why does the global variable `tcp_do_rfc1323` exist?

In [Figure 25.28](#), which did not use the timestamp option, the RTT estimators are updated eight times.

**26.4** If the timestamp option had been used in this example, how many times would the RTT estimators have been updated?

In [Figure 26.23](#) `bcopy` is called to store the received MSS in the variable `mss`. Why not cast the pointer to `opt[2]` into a pointer to an unsigned short and perform an assignment?

After [Figure 26.29](#) we described a bug in the code, which can cause a bogus urgent offset to be sent.  
**26.6** Propose a solution. (*Hint:* What is the largest amount of TCP data that can be sent in a segment?)

With Figure 26.32 we mentioned that an error of ENOBUFS is not returned to the process because (1) if the discarded segment contained data, the retransmission timer will expire and the data will be

**26.7** retransmitted, or (2) if the discarded segment was an ACK-only segment, the other end will retransmit its data when it doesn't receive the ACK. What if the discarded segment contains an RST?

**26.8** Explain the settings of the PSH flag in Figure 20.3 of Volume 1.

**26.9** Why does Figure 26.36 use the value of ip\_defttl for the TTL, while Figure 26.32 uses the value in the PCB?

Describe what happens with the mbuf allocated in Figure 26.25

**26.10** when IP options are specified by the process for the TCP connection. Implement a better solution.

`tcp_output` is a long function (about 500 lines, including comments), which can appear to be inefficient. But lots of the code handles special cases. Assume the function is called with a full-sized segment ready to

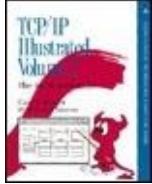
**26.11** be sent, and no special cases: no IP options and no special flags such as SYN, FIN, or URG. About how many lines of C code are actually executed? How many functions are called before the segment is passed to `ip_output`?

In the example at the end of [Section 26.3](#) in which the application did a write of 100 bytes followed by a write of 50 bytes, would anything change if the

**26.12** application called `writev` once for both buffers, instead of calling `write`

twice? Does anything change with writev if the two buffer lengths are 200 and 300, instead of 100 and 50?

The timestamp that is sent in the timestamp option is taken from the **26.13** global tcp\_now, which is incremented every 500 ms. Modify TCP to use a higher resolution timestamp value.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 27. TCP Functions

[Section 27.1. Introduction](#)

[Section 27.2. tcp\\_drain Function](#)

[Section 27.3. tcp\\_drop Function](#)

[Section 27.4. tcp\\_close Function](#)

[Section 27.5. tcp\\_mss Function](#)

[Section 27.6. tcp\\_ctlinput Function](#)

[Section 27.7. tcp\\_notify Function](#)

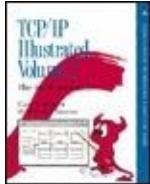
[Section 27.8. tcp\\_quench Function](#)

[Section 27.9. TCP\\_REASS Macro and  
tcp\\_reass Function](#)

[Section 27.10. tcp\\_trace Function](#)

[Section 27.11. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.1 Introduction

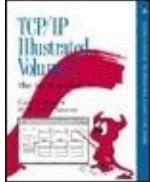
This chapter presents numerous TCP functions that we need to cover before discussing TCP input in the next two chapters:

- `tcp_drain` is the protocol's drain function, called when the kernel is out of mbufs. It does nothing.
- `tcp_drop` aborts a connection by sending an RST.
- `tcp_close` performs the normal TCP connection termination: send a FIN and wait for the four-way exchange to complete. Section 18.2 of Volume 1 talks about the four packets that are

exchanged when a connection is closed.

- `tcp_mss` processes a received MSS option and calculates the MSS to announce when TCP sends an MSS option of its own.
- `tcp_ctlinput` is called when an ICMP error is received in response to a TCP segment, and it calls `tcp_notify` to process the ICMP error. `tcp_quench` is a special case function that handles ICMP source quench errors.
- The `TCP_REASS` macro and the `tcp_reass` function manipulate segments on TCP's reassembly queue for a given connection. This queue handles the receipt of out-of-order segments, some of which might overlap.
- `tcp_trace` adds records to the kernel's circular debug buffer for TCP (the `SO_DEBUG` socket option) that can be printed with the `trpt(8)` program.

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.2 tcp\_drain Function

The simplest of all the TCP functions is `tcp_drain`. It is the protocol's `pr_drain` function, called by `m_reclaim` when the kernel runs out of mbufs. We saw in [Figure 10.32](#) that `ip_drain` discards all the fragments on its reassembly queue, and UDP doesn't define a drain function. Although TCP holds onto mbufssegments that have arrived out of order, but within the receive window for the socketthe Net/3 implementation of TCP does not discard these pending mbufs if the kernel runs out of space. Instead, `tcp_drain` does nothing, on the assumption that a received (but out-of-order) TCP segment is "more important" than an IP fragment.

---

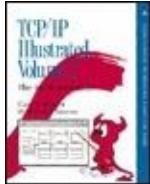
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.3 tcp\_drop Function

tcp\_drop is called from numerous places to drop a connection by sending an RST and to report an error to the process. This differs from closing a connection (the tcp\_disconnect function), which sends a FIN to the other end and follows the connection termination steps in the state transition diagram.

[Figure 27.1](#) shows the seven places where tcp\_drop is called and the errno argument.

**Figure 27.1. Calls to tcp\_drop and errno argument.**

Function	errno	Description
tcp_input	ENOBUFS	SYN arrives on listening socket, but kernel out of mbufs for t_template.
tcp_input	ECONNREFUSED	RST received in response to SYN.
tcp_input	ECONNRESET	RST received on existing connection.
tcp_timers	ETIMEDOUT	Retransmission timer has expired 13 times in a row with no ACK from other end (Figure 25.25).
tcp_timers	ETIMEDOUT	Connection-establishment timer has expired (Figure 25.15), or keepalive timer has expired with no response to nine consecutive probes (Figure 25.17)
tcp_usrreq	ECONNABORTED	PRU_ABORT request.
tcp_usrreq	0	Socket closed and SO_LINGER socket option set with linger time of 0.

Figure 27.2 shows the tcp\_drop function.

## Figure 27.2. tcp\_drop function.

```

202 struct tcpcb *
203 tcp_drop(tp, errno)
204 struct tcpcb *tp;
205 int     errno;
206 {
207     struct socket *so = tp->t_inpcb->inp_socket;
208     if (TCPS_HAVERCVDSYN(tp->t_state)) {
209         tp->t_state = TCPS_CLOSED;
210         (void) tcp_output(tp);
211         tcpstat.tcps_drops++;
212     } else
213         tcpstat.tcps_conndrops++;
214     if (errno == ETIMEDOUT && tp->t_softerror)
215         errno = tp->t_softerror;
216     so->so_error = errno;
217     return (tcp_close(tp));
218 }
```

202-213

If TCP has received a SYN, the connection is synchronized and an RST must be sent to the other end. This is done by setting the state to CLOSED and calling tcp\_output. In Figure 24.16 the value of tcp\_outflags for the CLOSED state includes

the RST flag.

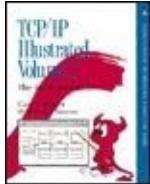
214-216

If the error is ETIMEDOUT but a soft error was received on the connection (e.g., EHOSTUNREACH), the soft error becomes the socket error, instead of the less specific ETIMEDOUT.

217

tcp\_close finishes closing the socket.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.4 `tcp_close` Function

`tcp_close` is normally called by `tcp_input` when the process has done a passive close and the ACK is received in the LAST\_ACK state, and by `tcp_timers` when the 2MSL timer expires and the socket moves from the TIME\_WAIT to CLOSED state. It is also called in other states, possibly after an error has occurred, as we saw in the previous section. It releases the memory occupied by the connection (the IP and TCP header template, the TCP control block, the Internet PCB, and any out-of-order segments remaining on the connection's reassembly queue) and updates the route characteristics.

We describe this function in three parts, the first two dealing with the route characteristics and the final part showing the release of resources.

## Route Characteristics

Nine variables are maintained in the `rt_metrics` structure ([Figure 18.26](#)), six of which are used by TCP. Eight of these can be examined and changed with the `route(8)` command (the ninth, `rmx_pktsent` is never used): these variables are shown in [Figure 27.3](#).

**Figure 27.3. Members of the `rt_metrics` structure used by TCP.**

<code>rt_metrics</code> member	saved by <code>tcp_close?</code>	used by <code>tcp_mss?</code>	<code>route(8)</code> modifier
<code>rmx_expire</code>			<code>-expire</code>
<code>rmx_hopcount</code>			<code>-hopcount</code>
<code>rmx_mtu</code>		•	<code>-mtu</code>
<code>rmx_recvpipe</code>		•	<code>-recvpipe</code>
<code>rmx_rtt</code>	•	•	<code>-rtt</code>
<code>rmx_rttvar</code>	•	•	<code>-rttvar</code>
<code>rmx_sendpipe</code>	•	•	<code>-sendpipe</code>
<code>rmx_sssthresh</code>	•	•	<code>-sssthresh</code>

Additionally, the `-lock` modifier can be used

with the route command to set the corresponding RTV\_xxx bit in the rmx\_locks member ([Figure 20.13](#)). Setting the RTV\_xxx bit tells the kernel not to update that metric.

When a TCP socket is closed, `tcp_close` updates three of the routing metrics—the smoothed RTT estimator, the smoothed mean deviation estimator, and the slow start threshold—but only if enough data was transferred on the connection to yield meaningful statistics and the variable is not locked.

[Figure 27.4](#) shows the first part of `tcp_close`.

**Figure 27.4. `tcp_close` function: update RTT and mean deviation.**

---

```

225 struct tcpcb *
226 tcp_close(tp)
227 struct tcpcb *tp;
228 {
229     struct tciphdr *t;
230     struct inpcb *inp = tp->t_inpcb;
231     struct socket *so = inp->inp_socket;
232     struct mbuf *m;
233     struct rtentry *rt;
234     /*
235      * If we sent enough data to get some meaningful characteristics,
236      * save them in the routing entry. 'Enough' is arbitrarily
237      * defined as the sendpipesize (default 8K) * 16. This would
238      * give us 16 rtt samples assuming we only get one sample per
239      * window (the usual case on a long haul net). 16 samples is
240      * enough for the srtt filter to converge to within 5% of the correct
241      * value; fewer samples and we could save a very bogus rtt.
242      */
243      * Don't update the default route's characteristics and don't
244      * update anything that the user *locked*.
245      */
246 if (SEQ_LT(tp->iss + so->so_snd.sb_hiwat * 16, tp->snd_max) &&
247     (rt = inp->inp_route.ro_rt) &&
248     ((struct sockaddr_in *) rt->sin_key(rt))>sin_addr.s_addr != INADDR_ANY) {
249     u_long i;
250
251     if ((rt->rt_rmx.rmx_locks & RTV_RTT) == 0) {
252         i = tp->t_srtt *
253             (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));
254         if (rt->rt_rmx.rmx_rtt && i)
255             /*
256              * filter this update to half the old & half
257              * the new values, converting scale.
258              * See route.h and tcp_var.h for a
259              * description of the scaling constants.
260              */
261         rt->rt_rmx.rmx_rtt =
262             (rt->rt_rmx.rmx_rtt + i) / 2;
263         else
264             rt->rt_rmx.rmx_rtt = i;
265     }
266     if ((rt->rt_rmx.rmx_locks & RTV_RTTVAR) == 0) {
267         i = tp->t_rttvar *
268             (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
269         if (rt->rt_rmx.rmx_rttvar && i)
270             rt->rt_rmx.rmx_rttvar =
271                 (rt->rt_rmx.rmx_rttvar + i) / 2;
272         else
273             rt->rt_rmx.rmx_rttvar = i;

```

---

tcp\_subr.c

## Check if enough data sent to update statistics

234-248

The default send buffer size is 8192 bytes

(`sb_hiwat`), so the first test is whether 131,072 bytes (16 full buffers) have been transferred across the connection. The initial send sequence number is compared to the maximum sequence number sent on the connection. Additionally the socket must have a cached route and that route cannot be the default route. (See [Exercise 19.2.](#))

Notice there is a small chance for an error in the first test, because of sequence number wrap, if the amount of data transferred is within  $N \times 2^{32}$  and  $N \times 2^{32} + 131072$ , for any  $N$  greater than 1. But few connections (today) transfer 4 gigabytes of data.

Despite the prevalence of default routes in the Internet, this information is still useful to maintain in the routing table. If a host continually exchanges data with another host (or network), even if a default route can be used, a host-specific or network-specific route can be entered into the routing table with the `route` command just to maintain this

information across connections. (See [Exercise 19.2.](#)) This information is lost when the system is rebooted.

250

The administrator can lock any of the variables from [Figure 27.3](#), preventing them from being updated by the kernel, so before modifying each variable this lock must be checked.

## Update RTT

251-264

`t_srtt` is stored as ticks  $\times$  8 ([Figure 25.19](#)) and `rmx_rtt` is stored as microseconds. So `t_srtt` is multiplied by 1,000,000 (RTM\_RTTUNIT) and then divided by 2 (ticks/second) times 8. If a value for `rmx_rtt` already exists, the new value is one-half the old value plus one-half the new value. Otherwise the new value is stored in `rmx_rtt`.

## Update mean deviation

265-273

The same algorithm is applied to the mean deviation estimator. It too is stored as microseconds, requiring a conversion from the t\_rttvar units of ticks x 4.

Figure 27.5 shows the next part of tcp\_close, which updates the slow start threshold for the route.

### Figure 27.5. tcp\_close function: update slow start threshold.

```
274     /*
275      * update the pipelimit (ssthresh) if it has been updated
276      * already or if a pipesize was specified & the threshhold
277      * got below half the pipesize. I.e., wait for bad news
278      * before we start updating, then update on both good
279      * and bad news.
280      */
281     if ((rt->rt_rmx.rmx_locks & RTV_SSTHRESH) == 0 &&
282         (i = tp->snd_ssthresh) && rt->rt_rmx.rmx_ssthresh ||
283         i < (rt->rt_rmx.rmx_sendpipe / 2)) {
284     /*
285      * convert the limit from user data bytes to
286      * packets then to packet data bytes.
287      */
288     i = (i + tp->t_maxseg / 2) / tp->t_maxseg;
289     if (i < 2)
290         i = 2;
291     i *= (u_long) (tp->t_maxseg + sizeof(struct tciphdr));
292     if (rt->rt_rmx.rmx_ssthresh)
293         rt->rt_rmx.rmx_ssthresh =
294             (rt->rt_rmx.rmx_ssthresh + i) / 2;
295     else
296         rt->rt_rmx.rmx_ssthresh = i;
297     }
298 }
```

274-283

The slow start threshold is updated only if  
(1) it has been updated already  
(`rmx_ssthresh` is nonzero) or (2)  
`rmx_sendpipe` is specified by the  
administrator and the new value of  
`snd_ssthresh` is less than one-half the  
value of `rmx_sendpipe`. As the comment in  
the code indicates, TCP does not update  
the value of `rmx_ssthresh` until it is forced  
to because of packet loss; from that point  
on it considers itself free to adjust the  
value either up or down.

284-290

The variable `snd_ssthresh` is maintained in  
bytes. The first conversion divides this  
variable by the MSS (`t_maxseg`), yielding  
the number of segments. The addition of  
one-half `t_maxseg` rounds the integer  
result. The lower bound on this result is  
two segments.

291-297

The size of the IP and TCP headers (40) is  
added to the MSS and multiplied by the  
number of segments. This value updates

`rmx_ssthresh`, using the same filtering as in [Figure 27.4](#) (one-half the old plus one-half the new).

## Resource Release

The final part of `tcp_close`, shown in [Figure 27.6](#), releases the memory resources held by the socket.

**Figure 27.6. `tcp_close` function: release connection resources.**

```
299  /* free the reassembly queue, if any */          tcp_subr.c
300  t = tp->seg_next;
301  while (t != (struct tciphdr *) tp) {
302      t = (struct tciphdr *) t->ti_next;
303      m = REASS_MBUF((struct tciphdr *) t->ti_prev);
304      remque(t->ti_prev);
305      m_free(m);
306  }
307  if (tp->t_template)
308      (void) m_free(dtom(tp->t_template));
309  free(tp, M_PCB);
310  inp->inp_ppcb = 0;
311  soisdisconnected(so);
312  /* clobber input pcb cache if we're closing the cached connection */
313  if (inp == tcp_last_inpcb)
314      tcp_last_inpcb = &tcb;
315  in_pcbdetach(inp);
316  tcpstat.tcps_closed++;
317  return ((struct tcpcb *) 0);
318 }
```

**Release any mbufs on reassembly queue**

299-306

If any segments are left on the connection's reassembly queue, they are discarded. This queue is for segments that arrive out of order but within the receive window. They are held in a reassembly queue until the required "earlier" segments are received, at which time they are reassembled and passed to the application in the correct order. We discuss this in more detail in [Section 27.9](#).

## **Release header template and TCP control block**

307-311

The template of the IP and TCP headers is released by `m_free` and the TCP control block is released by `free`. `soisdisconnected` marks the socket as disconnected.

## **Release PCB**

312-318

If the Internet PCB for this socket is the one currently cached by TCP, the cache is marked as empty by setting `tcp_last_inpcb` to the head of TCP's PCB list. The PCB is then detached, which releases the memory used by the PCB.

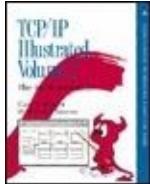
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.5 tcp\_mss Function

The `tcp_mss` function is called from two other functions:

- 1. from `tcp_output`, when a SYN segment is being sent, to include an MSS option, and**
  - from `tcp_input`, when an MSS option is received in a SYN segment.

The `tcp_mss` function checks for a cached route to the destination and calculates the MSS to use for this connection.

[Figure 27.7](#) shows the first part of `tcp_mss`, which acquires a route to the destination if one is not already held by

the PCB.

## Figure 27.7. `tcp_mss` function: acquire a route if one is not held by the PCB.

```
-----tcp_input.c
1391 int
1392 tcp_mss(tp, offer)
1393 struct tcpcb *tp;
1394 u_int offer;
1395 {
1396     struct route *ro;
1397     struct rtentry *rt;
1398     struct ifnet *ifp;
1399     int rtt, mss;
1400     u_long bufsize;
1401     struct inpcb *inp;
1402     struct socket *so;
1403     extern int tcp_mssdflt;
1404
1405     inp = tp->t_inpcb;
1406     ro = &inp->inp_route;
1407
1408     if ((rt = ro->ro_rt) == (struct rtentry *) 0) {
1409         /* No route yet, so try to acquire one */
1410         if (inp->inp_faddr.s_addr != INADDR_ANY) {
1411             ro->ro_dst.sa_family = AF_INET;
1412             ro->ro_dst.sa_len = sizeof(ro->ro_dst);
1413             ((struct sockaddr_in *) &ro->ro_dst)->sin_addr =
1414                 inp->inp_faddr;
1415             rtalloc(ro);
1416         }
1417         if ((rt = ro->ro_rt) == (struct rtentry *) 0)
1418             return (tcp_mssdflt);
1419     }
1420     ifp = rt->rt_ifp;
1421     so = inp->inp_socket;
-----tcp_input.c
```

## Acquire a route if necessary

1391-1417

If the socket does not have a cached route, `rtalloc` acquires one. The interface pointer associated with the outgoing route

is saved in ifp. Knowing the outgoing interface is important, since its associated MTU can affect the MSS announced by TCP. If a route is not acquired, the default of 512 (tcp\_mssdflt) is returned immediately.

The next part of tcp\_mss, shown in [Figure 27.8](#), checks whether the route has metrics associated with it; if so, the variables t\_rttmin, t\_srtt, and t\_rttvar can be initialized from the metrics.

**Figure 27.8. `tcp_mss` function: check if the route has an associated RTT metric.**

---

```

1420  /*
1421   * While we're here, check if there's an initial rtt
1422   * or rttvar. Convert from the route-table units
1423   * to scaled multiples of the slow timeout timer.
1424   */
1425  if (tp->t_srtt == 0 && (rtt = rt->rt_rmx.rmx_rtt)) {
1426      /*
1427       * XXX the lock bit for RTT indicates that the value
1428       * is also a minimum value; this is subject to time.
1429       */
1430      if (rt->rt_rmx.rmx_locks & RTV_RTT)
1431          tp->t_rttmin = rtt / (RTM_RTTUNIT / PR_SLOWHZ);
1432      tp->t_srtt = rtt / (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTT_SCALE));

1433      if (rt->rt_rmx.rmx_rttvar)
1434          tp->t_rttvar = rt->rt_rmx.rmx_rttvar /
1435                          (RTM_RTTUNIT / (PR_SLOWHZ * TCP_RTTVAR_SCALE));
1436      else
1437          /* default variation is +- 1 rtt */
1438          tp->t_rttvar =
1439              tp->t_srtt * TCP_RTTVAR_SCALE / TCP_RTT_SCALE;

1440      TCPT_RANGESET(tp->t_rxcur,
1441                     ((tp->t_srtt >> 2) + tp->t_rttvar) >> 1,
1442                     tp->t_rttmin, TCPTV_REXMTMAX);
1443  }

```

---

tcp\_input.c

## Initialize smoothed RTT estimator

*1420-1432*

If there are no RTT measurements yet for the connection (`t_srtt` is 0) and `rmx_rtt` is nonzero, the latter initializes the smoothed RTT estimator `t_srtt`. If the `RTV_RTT` bit in the routing metric lock flag is set, it indicates that `rmx_rtt` should also be used to initialize the minimum RTT for this connection (`t_rttmin`). We saw that `tcp_newtcpcb` initializes `t_rttmin` to 2 ticks.

`rmx_rtt` (in units of microseconds) is

converted to t\_srtt (in units of ticks x 8). This is the reverse of the conversion done in [Figure 27.4](#). Notice that t\_rttmin is set to one-eighth the value of t\_srtt, since the former is not divided by the scale factor TCP\_RTT\_SCALE.

## Initialize smoothed mean deviation estimator

1433-1439

If the stored value of rmx\_rttvar is nonzero, it is converted from units of microseconds into ticks x 4 and stored in t\_rttvar. But if the value is 0, t\_rttvar is set to t\_rtt, that is, the variation is set to the mean. This defaults the variation to  $\pm 1$  RTT. Since the units of the former are ticks x 4 and the units of the latter are ticks x 8, the value of t\_srtt is converted accordingly.

## Calculate initial RTO

1440-1442

The current *RTO* is calculated and stored in *t\_rxcur*, using the unscaled equation

$$RTO = srtt + 2 \times rttvar$$

A multiplier of 2, instead of 4, is used to calculate the first *RTO*. This is the same equation that was used in [Figure 25.21](#). Substituting the scaling relationships we get

$$\begin{aligned} RTO &= \frac{t\_srtt}{8} + 2 \times \frac{t\_rttvar}{4} \\ &= \frac{\frac{t\_srtt}{4} + t\_rttvar}{2} \end{aligned}$$

which is the second argument to *TCPT\_RANGESET*.

The next part of *tcp\_mss*, shown in [Figure 27.9](#), calculates the MSS.

**Figure 27.9. *tcp\_mss* function: calculate MSS.**

```
1444     /*  
1445      * if there's an mtu associated with the route, use it  
1446      */  
1447      if (rt->rt_rmx.rmx_mtu)  
1448          mss = rt->rt_rmx.rmx_mtu - sizeof(struct tciphdr);  
1449      else {  
1450          mss = ifp->if_mtu - sizeof(struct tciphdr);  
1451 #if (MCLBYTES & (MCLBYTES - 1)) == 0  
1452         if (mss > MCLBYTES)  
1453             mss &= ~(MCLBYTES - 1);  
1454 #else  
1455         if (mss > MCLBYTES)  
1456             mss = mss / MCLBYTES * MCLBYTES;  
1457#endif  
1458         if (!in_localaddr(inp->inp_faddr))  
1459             mss = min(mss, tcp_mssdflt);  
1460     }  
-----tcp_input.c
```

## Use MSS from routing table MTU

**1444-1450**

If the MTU is set in the routing table, mss is set to that value. Otherwise mss starts at the value of the outgoing interface MTU minus 40 (the default size of the IP and TCP headers). For an Ethernet, mss would start at 1460.

## Round MSS down to multiple of MCLBYTES

**1451-1457**

The goal of these lines of code is to reduce the value of mss to the next-lower multiple

of the mbuf cluster size, if mss exceeds MCLBYTES. If the value of MCLBYTES (typically 1024 or 2048) logically ANDed with the value minus 1 equals 0, then MCLBYTES is a power of 2. For example, 1024 (0x400) logically ANDed with 1023 (0x3ff) is 0.

The value of mss is reduced to the next-lower multiple of MCLBYTES by clearing the appropriate number of low-order bits: if the cluster size is 1024, logically ANDing mss with the one's complement of 1023 (0xfffffc00) clears the low-order 10 bits. For an Ethernet, this reduces mss from 1460 to 1024. If the cluster size is 2048, logically ANDing mss with the one's complement of 2047 (0xffff8000) clears the low-order 11 bits. For a token ring with an MTU of 4464, this reduces the value of mss from 4424 to 4096. If MCLBYTES is not a power of 2, the rounding down to the next-lower multiple of MCLBYTES is done with an integer division followed by a multiplication.

## Check if destination local or nonlocal

## 1458-1459

If the foreign IP address is not local (in\_localaddr returns 0), and if mss is greater than 512 (tcp\_mssdflt), it is set to 512.

Whether an IP address is "local" or not depends on the value of the global subnetsarelocal, which is initialized from the symbol SUBNETSARELOCAL when the kernel is compiled. The default value is 1, meaning that an IP address with the same network ID as one of the host's interfaces is considered local. If the value is 0, an IP address must have the same network ID and the same subnet ID as one of the host's interfaces to be considered local.

This minimization for nonlocal hosts is an attempt to avoid fragmentation across wide-area networks. It is a historical artifact from the ARPANET when the MTU across most WAN links was 1006. As discussed in Section 11.7 of Volume 1, most WANs today support

an MTU of 1500 or greater. See also the discussion of the path MTU discovery feature (RFC 1191 [ [Mogul and Deering 1990](#)]), in Section 24.2 of Volume 1. Net/3 does not support path MTU discovery.

The final part of `tcp_mss` is shown in Figure 27.10.

**Figure 27.10. `tcp_mss` function:  
complete processing.**

---

```

1461  /*
1462   * The current mss, t_maxseg, was initialized to the default value
1463   * of 512 (tcp_mssdflt) by tcp_newtcpcb().
1464   * If we compute a smaller value, reduce the current mss.
1465   * If we compute a larger value, return it for use in sending
1466   * a max seg size option, but don't store it for use
1467   * unless we received an offer at least that large from peer.
1468   * However, do not accept offers under 32 bytes.
1469   */
1470 if (offer)
1471     mss = min(mss, offer);
1472 mss = max(mss, 32);           /* sanity */
1473 if (mss < tp->t_maxseg || offer != 0) {
1474     /*
1475      * If there's a pipesize, change the socket buffer
1476      * to that size. Make the socket buffers an integral
1477      * number of mss units; if the mss is larger than
1478      * the socket buffer, decrease the mss.
1479      */
1480     if ((bufsize = rt->rt_rmx.rmx_sendpipe) == 0)
1481         bufsize = so->so_snd.sb_hiwat;
1482     if (bufsize < mss)
1483         mss = bufsize;
1484     else {
1485         bufsize = roundup(bufsize, mss);
1486         if (bufsize > sb_max)
1487             bufsize = sb_max;
1488         (void) sbreserve(&so->so_snd, bufsize);
1489     }
1490     tp->t_maxseg = mss;
1491     if ((bufsize = rt->rt_rmx.rmx_recvpipe) == 0)
1492         bufsize = so->so_rcv.sb_hiwat;
1493     if (bufsize > mss) {
1494         bufsize = roundup(bufsize, mss);
1495         if (bufsize > sb_max)
1496             bufsize = sb_max;
1497         (void) sbreserve(&so->so_rcv, bufsize);
1498     }
1499 }
1500 tp->snd_cwnd = mss;
1501 if (rt->rt_rmx.rmx_ssthresh) {
1502     /*
1503      * There's some sort of gateway or interface
1504      * buffer limit on the path. Use this to set
1505      * the slow start threshhold, but set the
1506      * threshold to no less than 2*mss.
1507      */
1508     tp->snd_ssthresh = max(2 * mss, rt->rt_rmx.rmx_ssthresh);
1509 }
1510 return (mss);
1511 }

```

---

tcp\_input.c

## Other end's MSS is upper bound

1461-1472

The argument offer is nonzero when this

function is called from `tcp_input`, and its value is the MSS advertised by the other end. If the value of `mss` is greater than the value advertised by the other end, it is set to the value of `offer`. For example, if the function calculates an `mss` of 1024 but the advertised value from the other end is 512, `mss` must be set to 512. Conversely, if `mss` is calculated as 536 (say the outgoing MTU is 576) and the other end advertises an MSS of 1460, TCP will use 536. TCP can always use a value less than the advertised MSS, but it can't exceed the advertised value. The argument `offer` is 0 when this function is called by `tcp_output` to send an MSS option. The value of `mss` is also lower-bounded by 32.

### 1473-1483

If the value of `mss` has decreased from the default set by `tcp_newtcpcb` in the variable `t_maxseg` (512), or if TCP is processing a received MSS option (`offer` is nonzero), the following steps occur. First, if the value of `rmx_sendpipe` has been stored for the route, its value will be used as the send buffer high-water mark ( [Figure 16.4](#)). If

the buffer size is less than mss, the smaller value is used. This should never happen unless the application explicitly sets the send buffer size to a small value, or the administrator sets rmx\_sendpipe to a small value, since the high-water mark of the send buffer defaults to 8192, larger than most values for the MSS.

## Round buffer sizes to multiple of MSS

### 1484-1489

The send buffer size is rounded up to the next integral multiple of the MSS, bounded by the value of sb\_max (262, 144 on Net/3, which is 256x1024). The socket's high-water mark is set by sbreserve. For example, the default high-water mark is 8192, but for a local TCP connection on an Ethernet with a cluster size of 2048 (i.e., an MSS of 1460) this code increases the high-water mark to 8760 (which is 6x1460). But for a nonlocal connection with an MSS of 512, the high-water mark is left at 8192.

*1490*

The value of `t_maxseg` is set, either because it decreased from the default (512) or because an MSS option was received from the other end.

*1491-1499*

The same logic just applied to the send buffer is also applied to the receive buffer.

## **Initialize congestion window and slow start threshold**

*1500-1509*

The value of the congestion window, `snd_cwnd`, is set to one segment. If the `rmx_ssthresh` value in the routing table is nonzero, the slow start threshold (`snd_ssthresh`) is set to that value, but the value must not be less than two segments.

*1510*

The value of `mss` is returned by the function. `tcp_input` ignores this value in

[Figure 28.10](#) (since it received an MSS from the other end), but `tcp_output` sends this value as the announced MSS in [Figure 26.23](#).

## Example

Let's go through an example of a TCP connection establishment and the operation of `tcp_mss`, since it can be called twice: once when the SYN is sent and once when a SYN is received with an MSS option.

### **1. The socket is created and `tcp_newtcpcb` sets `t_maxseg` to 512.**

- The process calls `connect`, and `tcp_output` calls `tcp_mss` with an offer argument of 0, to include an MSS option with the SYN. Assuming a local destination, an Ethernet LAN, and an mbuf cluster size of 2048, `mss` is set to 1460 by the code in [Figure 27.9](#). Since offer is 0, [Figure 27.10](#) leaves the value as 1460 and this is the function's return value. The

buffer sizes aren't modified, since 1460 is larger than the default (512) and a value hasn't been received from the other end yet. `tcp_output` sends an MSS option announcing a value of 1460.

- The other end replies with its SYN, announcing an MSS of 1024. `tcp_input` calls `tcp_mss` with an offer argument of 1024. The logic in [Figure 27.9](#) still yields a value of 1460 for `mss`, but the call to `min` at the beginning of [Figure 27.10](#) reduces this to 1024. Since the value of `offer` is nonzero, the buffer sizes are rounded up to the next integral multiple of 1024 (i.e., they're left at 8192). `t_maxseg` is set to 1024.

It might appear that the logic of `tcp_mss` is flawed: TCP announces an MSS of 1460 but receives an MSS of 1024 from the other end. While TCP is restricted to sending 1024-byte segments, the other end is free to send 1460-byte segments. We might think that the send buffer should be a multiple of 1024, but the receive buffer should be a multiple of 1460. Yet the

code in [Figure 27.10](#) sets both buffer sizes based on the *received* MSS. The reasoning is that even if TCP announces an MSS of 1460, since it receives an MSS of 1024 from the other end, the other end probably won't send 1460-byte segments, but will restrict itself to 1024-byte segments.

---

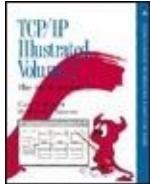
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.6 tcp\_ctlinput Function

Recall from [Figure 22.32](#) that `tcp_ctlinput` processes five types of ICMP errors: destination unreachable, parameter problem, source quench, time exceeded, and redirects. All redirects are passed to both TCP and UDP. For the other four errors, `tcp_ctlinput` is called only if a TCP segment caused the error.

`tcp_ctlinput` is shown in [Figure 27.11](#). It is similar to `udp_ctlinput`, shown in [Figure 23.30](#).

**Figure 27.11. `tcp_ctlinput` function.**

---

```

355 void
356 tcp_ctlinput(cmd, sa, ip)
357 int     cmd;
358 struct sockaddr *sa;
359 struct ip *ip;
360 {
361     struct tcphdr *th;
362     extern struct in_addr zeroin_addr;
363     extern u_char inetctllerrmap[];
364     void    (*notify) (struct inpcb *, int) = tcp_notify;

365     if (cmd == PRC_QUENCH)
366         notify = tcp_quench;
367     else if (!PRC_IS_REDIRECT(cmd) &&
368             ((unsigned) cmd > PRC_NCMDS || inetctllerrmap[cmd] == 0))
369         return;
370     if (ip) {
371         th = (struct tcphdr *) ((caddr_t) ip + (ip->ip_hl << 2));
372         in_pcbnotify(&tcb, sa, th->th_dport, ip->ip_src, th->th_sport,
373                      cmd, notify);
374     } else
375         in_pcbnotify(&tcb, sa, 0, zeroin_addr, 0, cmd, notify);
376 }

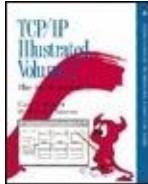
```

---

tcp\_subr.c

## 365-366

The only difference in the logic from `udp_ctlinput` is how an ICMP source quench error is handled. UDP ignores these errors since the `PRC_QUENCH` entry of `inetctllerrmap` is 0. TCP explicitly checks for this error, changing the `notify` function from its default of `tcp_notify` to `tcp_quench`.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.7 tcp\_notify Function

`tcp_notify` is called by `tcp_ctlinput` to handle destination unreachable, parameter problem, time exceeded, and redirect errors. This function is more complicated than its UDP counterpart, since TCP must intelligently handle soft errors for an established connection. [Figure 27.12](#) shows the `tcp_notify` function.

**Figure 27.12. `tcp_notify` function.**

```
328 void
329 tcp_notify(inp, error)
330 struct inpcb *inp;
331 int     error;
332 {
333     struct tcpcb *tp = (struct tcpcb *) inp->inp_ppcb;
334     struct socket *so = inp->inp_socket;

335     /*
336      * Ignore some errors if we are hooked up.
337      * If connection hasn't completed, has retransmitted several times,
338      * and receives a second error, give up now. This is better
339      * than waiting a long time to establish a connection that
340      * can never complete.
341     */
342     if (tp->t_state == TCPS_ESTABLISHED &&
343         (error == EHOSTUNREACH || error == ENETUNREACH ||
344          error == EHOSTDOWN)) {
345         return;
346     } else if (tp->t_state < TCPS_ESTABLISHED && tp->t_rxtshift > 3 &&
347                tp->t_softerror)
348         so->so_error = error;
349     else
350         tp->t_softerror = error;
351     wakeup((caddr_t) & so->so_timeo);
352     sorwakeups(so);
353     sowwakeups(so);
354 }
```

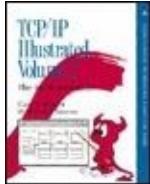
tcp\_subr.c

## 328-345

If the connection is ESTABLISHED, the errors EHOSTUNREACH, ENETUNREACH, and EHOSTDOWN are ignored.

This handling of these three errors is new with 4.4BSD. Net/2 and earlier releases recorded these errors in the connection's soft error variable (*t\_softerror*), and the error was reported to the process should the connection eventually fail. Recall that *tcp\_xmit\_timer* resets this variable to 0 when an ACK is received for a segment that hasn't been retransmitted.

If the connection is not yet established, TCP has retransmitted the current segment four or more times, and an error has already been recorded in `t_softerror`, the current error is recorded in the socket's `so_error` variable. By setting this socket variable, the socket becomes readable and writable if the process calls `select`. Otherwise the current error is just saved in `t_softerror`. We saw that `tcp_drop` sets the socket error to this saved value if the connection is subsequently dropped because of a timeout. Any processes waiting to receive or send on the socket are then awakened to receive the error.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.8 tcp\_quench Function

tcp\_quench, which is shown in [Figure 27.13](#), is called by tcp\_ctlinput when a source quench is received for the connection, and by tcp\_output ([Figure 26.32](#)) when ip\_output returns ENOBUFS.

**Figure 27.13. tcp\_quench function.**

```
381 void                                         tcp_subr.c
382 tcp_quench(inp, errno)
383 struct inpcb *inp;
384 int      errno;
385 {
386     struct tcpcb *tp = intotcpcb(inp);
387     if (tp)
388         tp->snd_cwnd = tp->t_maxseg;
389 }
```

The congestion window is set to one

segment, causing slow start to take over. The slow start threshold is not changed (as it is when `tcp_timers` handles a retransmission timeout), so the window will open up exponentially until `snd_ssthresh` is reached, or congestion occurs.

---

## Chapter 27. TCP Functions

---

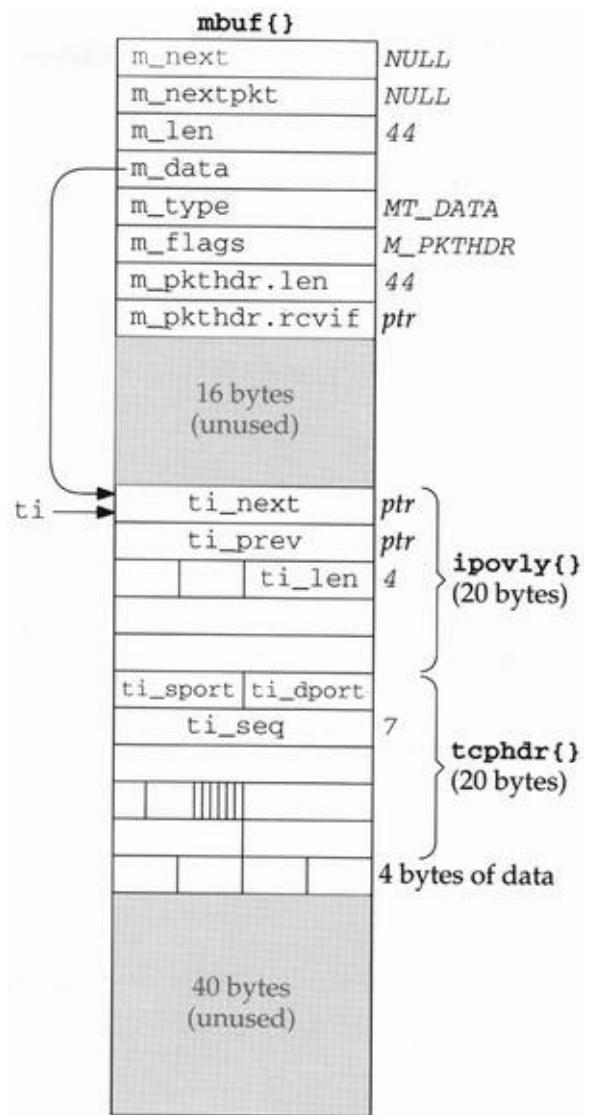
### 27.9 TCP\_REASS Macro and tcp\_reass() Function

TCP segments can arrive out of order, and it is the responsibility of the receiver to place the misordered segments into the correct sequence. For example, if a receiver advertises byte number 0 as the next expected byte, and receives bytes 01023 (an in-order segment) followed by 2048-3071, this second segment is out of order. The receiver must identify the out-of-order segment if it is within the received window. It then places the segment on the reassembly list for the connection. The receiver waits for the missing segment to arrive (with bytes 1024-2047). At this time it can acknowledge bytes 1024-3071 and proceed with the process. In this section we examine the code for the TCP reassembly queue, before discussing tcp\_input() in the next chapter.

If we assume that a single mbuf contains the IP header and 4 bytes of TCP data (recall the left half of Figure 27.13), then we have the arrangement shown in Figure 27.14. The figure shows a sequence of four mbufs, each containing 4 bytes of TCP data. The first mbuf has an offset of 0, the second has an offset of 4, the third has an offset of 8, and the fourth has an offset of 12. The total length of each mbuf is 16 bytes, which includes the 4 bytes of TCP data and the 12 bytes of IP header.

bytes are sequence numbers 7, 8, 9, and 10.

**Figure 27.14. Example mbuf with IP and TCP data.**



The `ipovly` and `tcpiphdr` structures form the `tcpip` showed in [Figure 24.12](#). We showed a picture c

[Figure 24.10](#). In [Figure 27.14](#) we show only the reassembly: `ti_next`, `ti_prev`, `ti_len`, `ti_sport`, `ti_dport`. The first two are pointers that form a doubly linked list of segments in order of arrival for a given connection. The header contains the control block for the connection: the `seg_next` pointer, which are the first two members of the structure. `ti_prev` pointers overlay the first 8 bytes of the segment header needed once the datagram reaches TCP. `ti_len` is the length of the data, and is calculated and stored by TCP before checksum.

## TCP\_REASS Macro

When data is received by `tcp_input`, the macro [Figure 27.15](#), is invoked to place the data onto the reassembly queue. This macro is called from or [Figure 29.22](#).

**Figure 27.15. TCP\_REASS macro: add data to connection.**

```

53 #define TCP_REASS(tp, ti, m, so, flags) ( \
54     if ((ti)->ti_seq == (tp)->rcv_nxt && \
55         (tp)->seg_next == (struct tciphdr *) (tp) && \
56         (tp)->t_state == TCPS_ESTABLISHED) { \
57         tp->t_flags |= TF_DELACK; \
58         (tp)->rcv_nxt += (ti)->ti_len; \
59         flags = (ti)->ti_flags & TH_FIN; \
60         tcpstat.tcpss_rcvpack++; \
61         tcpstat.tcpss_rcvbyte += (ti)->ti_len; \
62         sbappend(&(so)->so_rcv, (m)); \
63         sorwakeup(so); \
64     } else { \
65         (flags) = tcp_reass((tp), (ti), (m)); \
66         tp->t_flags |= TF_ACKNOW; \
67     } \
68 }

```

tcp\_input.c

## 54-63

tp is a pointer to the TCP control block for the connection, and  
 ti is a pointer to the tciphdr structure for the received segment.  
 The following three conditions are all true:

- 1. this segment is in-order (the sequence number matches the next expected sequence number for rcv\_nxt), and**

- the reassembly queue for the connection is empty (it's in the TCP control block itself, not some mbuf), and
- the connection is ESTABLISHED,

Under these conditions, the following steps take place: a delayed ACK is updated with the amount of data in the segment, the sequence number is set to TH\_FIN if the FIN flag is set in the TCP header, two statistics are updated, the data is appended to the receive buffer, and any receiving processes waiting for

awakened.

The reason all three conditions must be true is out of order, it must be placed onto the connection queue, and the "preceding" segments must be received and passed to the process. Second, even if the data is out-of-order data already on the reassembly queue, it must be checked to see if that the new segment might fill a hole, allowing the new segment and one or more segments on the queue to all be passed to the process. Third, it is OK for data to arrive with a connection, but that data cannot be passed to the process until the connection is ESTABLISHED. Any such connection is added to the reassembly queue when it arrives.

64-67

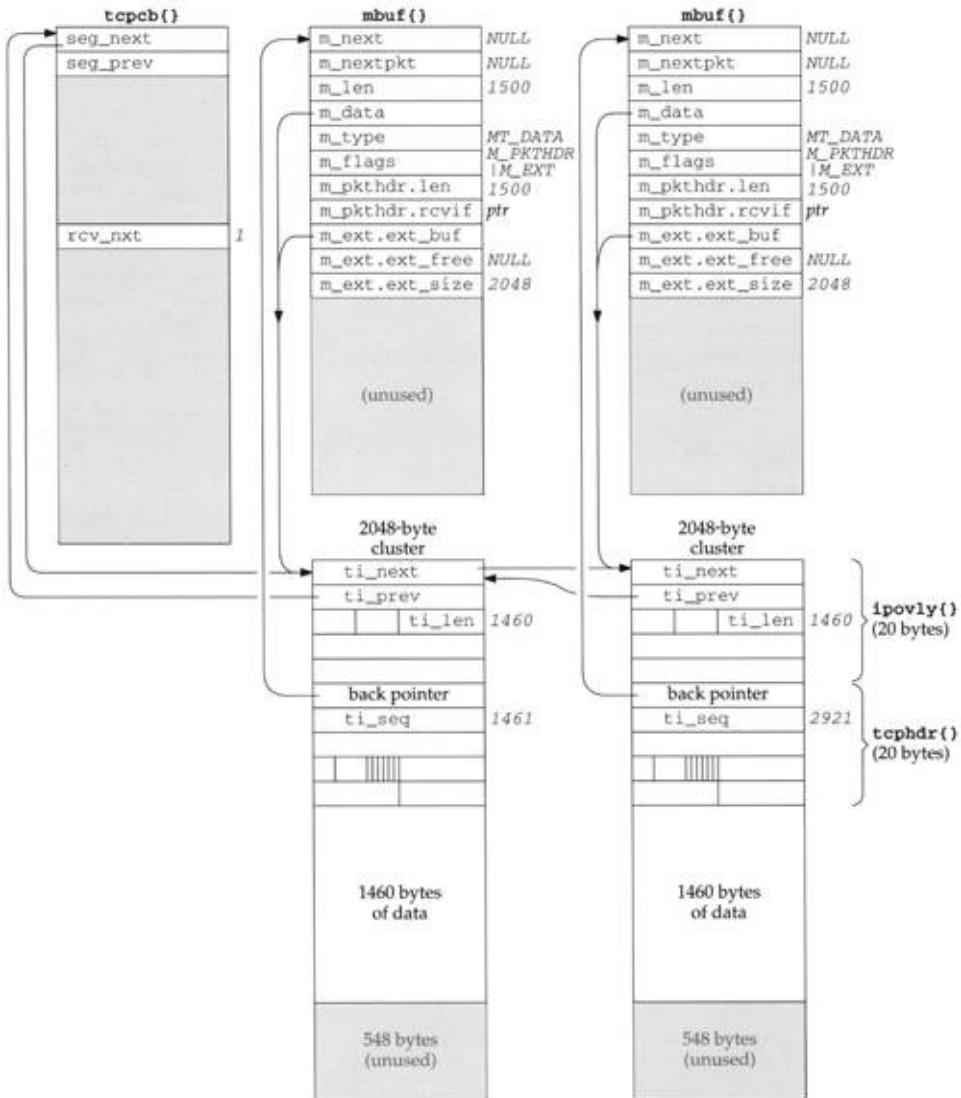
If these three conditions are not all true, the TCP code calls the function `tcp_reass` to add the segment to the reassembly queue. Since the segment is either out of order, or the sequence number is less than the sequence number of the previous segment from previously received out-of-order segments, the segment is not scheduled. One important feature of TCP is that it generates an immediate ACK when an out-of-order segment is received. This aids the *fast retransmit* algorithm ( [Section 27.14](#) ).

Before looking at the code for the `tcp_reass` function, let's explain what's done with the two port numbers in [Figure 27.14](#), `ti_sport` and `ti_dport`. Once the TCP connection is located and `tcp_reass` is called, these two port numbers are used to look up the connection information in the `tcp_table`.

needed. Therefore, when a TCP segment is placed on the queue, the address of the corresponding mbuf and port numbers. In [Figure 27.14](#) this isn't needed because the headers are in the data portion of the mbuf, so they can be copied directly. But recalling our discussion of `m_pullup` in [Section 2.10](#), we see that if the headers are in a cluster (as in [Figure 2.16](#), which shows a full-sized TCP segment), the `dtom` macro does not copy them. Instead, it copies the data portion. In that section that TCP stores its own back pointer to the mbuf, and that back pointer is stored over the port numbers.

[Figure 27.16](#) shows an example of this technique. It shows two segments for a connection, each segment stored in a mbuf. The head of the doubly linked list of out-of-order segments is a member of the control block for this connection. Note that we don't show the `seg_prev` pointer and the `ti_segment` on the list.

**Figure 27.16. Two out-of-order TCP segments in clusters.**



The next expected sequence number is 1 (rcv\_nxt was lost). The next two segments have containing bytes 1461-4380, but they are out of order and were placed into clusters by m\_devget, as shown next.

The first 32 bits of the TCP header contain a back pointer to the corresponding mbuf. This back pointer is used in the diagram shown next.

## tcp\_reass Function

Figure 27.17 shows the first part of the tcp\_reass function. The arguments are: tp, a pointer to the TCP control segment; ti, a pointer to the IP and TCP header segment; and m, a pointer to the mbuf chain for the data. As mentioned earlier, ti can point into the data pointed to by m, or ti can point into a cluster.

**Figure 27.17. tcp\_reass function**

```
69 int  
70 tcp_reass(tp, ti, m)  
71 struct tcpcb *tp;  
72 struct tciphdr *ti;  
73 struct mbuf *m;  
74 {  
75     struct tciphdr *q;  
76     struct socket *so = tp->t_inpcb->inp_socket;  
77     int      flags;  
78     /*  
79     * Call with ti==0 after become established to  
80     * force pre-ESTABLISHED data up to user socket.  
81     */  
82     if (ti == 0)  
83         goto present;  
84     /*  
85     * Find a segment that begins after this one does.  
86     */  
87     for (q = tp->seg_next; q != (struct tciphdr *) tp;  
88         q = (struct tciphdr *) q->ti_next)  
89         if (SEQ_GT(q->ti_seq, ti->ti_seq))  
90             break;
```

69-83

We'll see that `tcp_input` calls `tcp_reass` with a retransmitted segment that is acknowledged (Figures 28.20 and 29.2). This

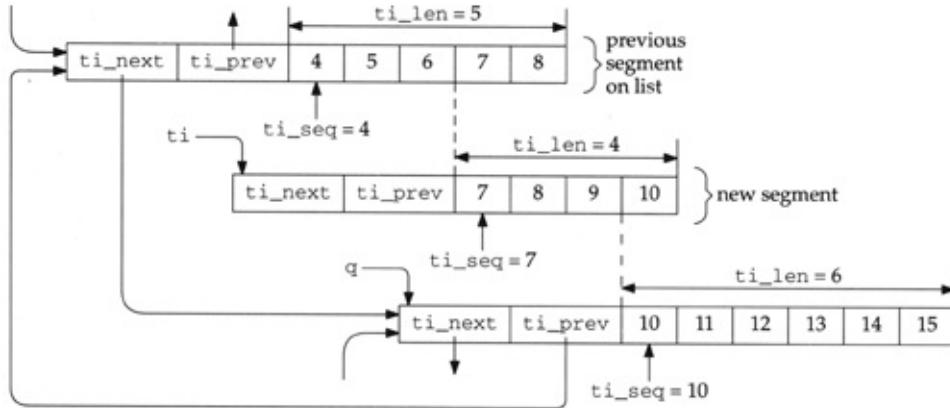
is now established, and any data that might have been queued (which `tcp_reass` had to queue earlier) can now be delivered to the application. Data that arrives with a SYN cannot be processed until the connection is established. The code for this is shown in Figure 27.23.

84-90

Go through the list of segments for this connection, starting at `seg_next`, to find the first one with a sequence number greater than the received sequence number (`ti_seq`). `N` is the entire body of the for loop.

Figure 27.18 shows an example with two out-of-order segments on the queue when a new segment arrives. We show the pointers `seg` pointing to the next segment on the list, the `seg_start` pointer to the start of the segment, and the `ti_next` pointer pointing to the next segment on the list. In this figure we also show the two pointers `ti_next` and `ti_start` pointing to the start of the segment. The `ti_start` pointer is the starting sequence number (`ti_seq`), the `ti_end` pointer is the ending sequence number, and the `ti_len` pointer is the length of the data bytes. With the exception of the first segment, each segment is probably in a single mbuf, as indicated by the `mbuf` pointer.

**Figure 27.18. Example of TCP reassembly of out-of-order segments.**



The next part of `tcp_reass` is shown in Figure 2

## Figure 27.19. `tcp_reass` function:

```

91  /*
92   * If there is a preceding segment, it may provide some of
93   * our data already. If so, drop the data from the incoming
94   * segment. If it provides all of our data, drop us.
95   */
96  if ((struct tciphdr *) q->ti_prev != (struct tciphdr *) tp) {
97      int i;
98      q = (struct tciphdr *) q->ti_prev;
99      /* conversion to int (in i) handles seq wraparound */
100     i = q->ti_seq + q->ti_len - ti->ti_seq;
101     if (i > 0) {
102         if (i >= ti->ti_len) {
103             tcpstat.tcp_rcvduppack++;
104             tcpstat.tcp_rcvdupbyte += ti->ti_len;
105             m_freem(m);
106             return (0);
107         }
108         m_adj(m, i);
109         ti->ti_len -= i;
110         ti->ti_seq += i;
111     }
112     q = (struct tciphdr *) (q->ti_next);
113 }
114 tcpstat.tcp_rcvoopack++;
115 tcpstat.tcp_rcvoobyte += ti->ti_len;
116 REASS_MBUF(ti) = m;           /* XXX */

```

91-107

If there is a segment before the one pointed to

overlap the new segment. The pointer q is moved to the next segment on the list (the one with bytes 4-8 in its sequence number). The number of bytes of overlap is calculated and stored in i:

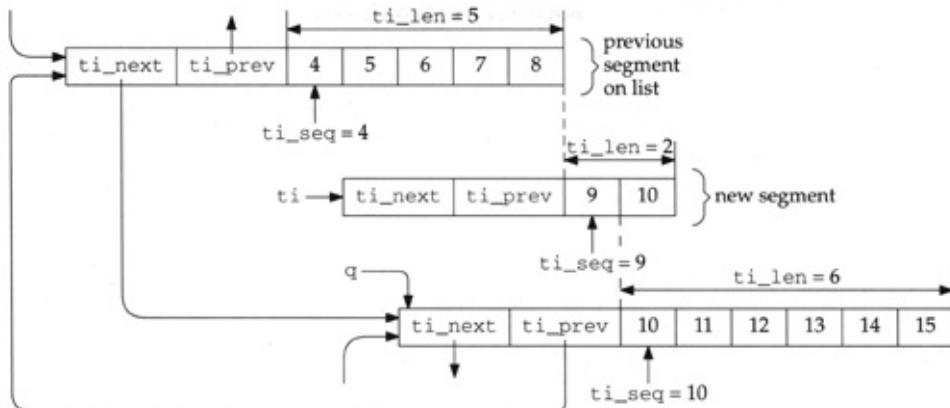
```
i = q->ti_seq + q->ti_len - ti->ti_seq  
= 4 + 5 - 7  
= 2
```

If i is greater than 0, there is overlap, as we have already calculated. The number of bytes of overlap in the previous segment is greater than or equal to the size of the new segment. The first i bytes in the new segment are already contained in the previous segment on the list. In this case the duplicate sequence number is discarded.

## 108-112

If there is only partial overlap (as there is in Figure 27.18), the function discards i bytes of data from the beginning of the new segment, and updates the sequence number and length of the new segment accordingly. q is moved to the next segment or to the end of the list, whichever comes first. Let's see our example at this point.

**Figure 27.20. Update of Figure 27.18 after bytes 4-7 are removed from new segment**



116

The address of the mbuf `m` is stored in the TCP and destination TCP ports. We mentioned earlier provides a back pointer from the TCP header to the TCP header is stored in a cluster, meaning that work. The macro `REASS_MBUF` is

```
#define REASS_MBUF(ti) (* (struct mbuf *)
```

`ti_t` is the `tcpiphdr` structure ( [Figure 24.12](#)) and the structure are the two 16-bit port numbers. [Figure 27.19](#) is because this hack assumes that bits occupied by the two port numbers.

The third part of `tcp_reass` is shown in [Figure 27.21](#). It handles the overlap from the next segment in the queue.

**Figure 27.21. `tcp_reass` function**

```

117  /*
118   * While we overlap succeeding segments trim them or,
119   * if they are completely covered, dequeue them.
120   */
121  while (q != (struct tcphdr *) tp) {
122      int i = (ti->ti_seq + ti->ti_len) - q->ti_seq;
123      if (i <= 0)
124          break;
125      if (i < q->ti_len) {
126          q->ti_seq += i;
127          q->ti_len -= i;
128          m_adj(REASS_MBUF(q), i);
129          break;
130      }
131      q = (struct tcphdr *) q->ti_next;
132      m = REASS_MBUF((struct tcphdr *) q->ti_prev);
133      remque(q->ti_prev);
134      m_free(m);
135  }
136  /*
137   * Stick new segment in its place.
138   */
139  insque(ti, q->ti_prev);

```

tcp\_input.c

## 117-135

If there is another segment on the list, the number between the new segment and that segment is example we have

$$\begin{aligned} i &= 9 + 2 - 10 \\ &= 1 \end{aligned}$$

since byte number 10 overlaps the two segments

Depending on the value of  $i$ , one of three conditions

### **1. If $i$ is less than or equal to 0, there is no overlap**

- If  $i$  is less than the number of bytes in the new segment, there is partial overlap and  $m\_adj$  removes the

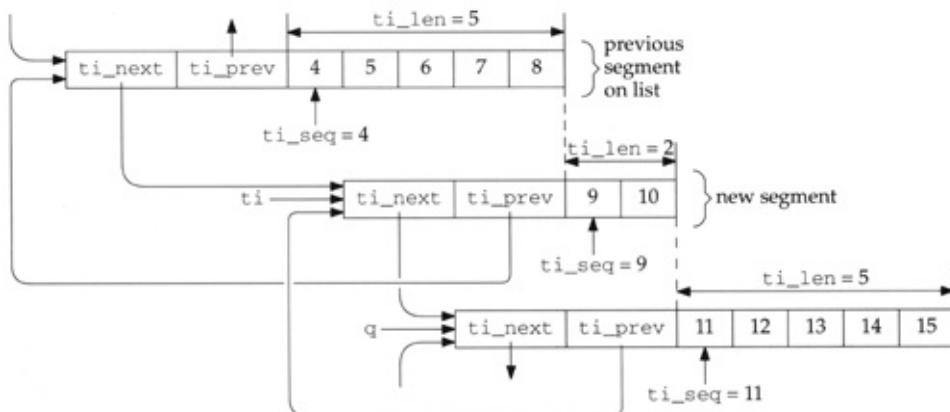
next segment on the list.

- If i is greater than or equal to the number of l segment, there is complete overlap and that ne deleted.

136-139

The new segment is inserted into the reasemb by insque. [Figure 27.22](#) shows the state of our

**Figure 27.22. Update of Figure 27.20 after rei bytes.**



[Figure 27.23](#) shows the final part of `tcp_reass`. process, if possible.

**Figure 27.23. `tcp_reass` function:**

```

140    present:
141    /*
142     * Present data to user, advancing rcv_nxt through
143     * completed sequence space.
144     */
145    if (TCPS_HAVERCVDSYN(tp->t_state) == 0)
146        return (0);
147    ti = tp->seg_next;
148    if (ti == (struct tciphdr *) tp || ti->ti_seq != tp->rcv_nxt)
149        return (0);
150    if (tp->t_state == TCPS_SYN_RECEIVED && ti->ti_len)
151        return (0);
152    do {
153        tp->rcv_nxt += ti->ti_len;
154        flags = ti->ti_flags & TH_FIN;
155        remque(ti);
156        m = REASS_MBUF(ti);
157        ti = (struct tciphdr *) ti->ti_next;
158        if (so->so_state & SS_CANTRCVMORE)
159            m_freem(m);
160        else
161            sbappend(&so->so_rcv, m);
162    } while (ti != (struct tciphdr *) tp && ti->ti_seq == tp->rcv_nxt);
163    sorwakeup(so);
164    return (flags);
165 }

```

tcp\_input.c

## 145-146

If the connection has not received a SYN (i.e., in SYN\_SENT state), data cannot be passed to the function returns. When this function is called by value of 0 is stored in the flags argument to the function. This is the side effect of clearing the FIN flag. We'll see this possibility when TCP\_REASS is invoked in [Figure 5-14](#). A received segment contains a SYN, FIN, and data (but valid).

## 147-149

ti starts at the first segment on the list. If the starting sequence number of the first segment does not equal the next receive sequence number,

function returns a value of 0. If the second connection has a hole in the received data starting with the next sequence number. For instance, in our example (Figure 2), if the sequence number 4 is missing from the list, but rcv\_nx is still 4, then bytes 4-8 are the first on the list but rcv\_nx is still 4, so bytes 4-15 cannot be passed to the application. A return of 0 turns off the FIN flag (if set), because there are still segments missing, so a received FIN can't be acknowledged.

## 150-151

If the state is SYN\_RCVD and the length of the function returns a value of 0. If both of these conditions are true, then the socket is a listening socket that has received in SYN-ACK segments. The data is left on the connection's queue, waiting for the handshake to complete.

## 152-164

This loop starts with the first segment on the list (if it is in order) and appends it to the socket's receive buffer. The index is incremented by the number of bytes in the segment. The loop continues until either the list is empty or when the sequence number of the segment on the list is out of order (i.e., there is a hole in the sequence space). When the loop terminates, the flags variable (the return value of the function) is 0 or TH\_FIN if the final segment placed in the socket's receive buffer had its FIN flag set or not.

After all the mbufs have been placed onto the socket, sorwakeup wakes any process waiting for data on the socket.

---

---

## Chapter 27. TCP Functions

---

### 27.10 tcp\_trace Function

In `tcp_output`, before sending a segment to IP, it calls `tcp_trace` in [Figure 26.32](#):

```
if (so->so_options & SO_DEBUG)
    tcp_trace(TA_OUTPUT,...);
```

This call adds a record to a circular buffer in the kernel's memory, which can be read by the `trpt(8)` program. Additionally, if the kernel is compiled with the `tcpconsdebug` option and if the variable `tcpconsdebug` is nonzero, information will be printed to the console.

Any process can set the `SO_DEBUG` socket option to request that information to be stored in the kernel's circular buffer. A user-space application can read this information from kernel memory (`/dev/kmem`) to fetch this information without requiring special privileges.

The `SO_DEBUG` socket option can be set for a socket using the `setsockopt` system call:

raw IP), but TCP is the only protocol that look

The information saved by the kernel is a `tcp_debug`.  
[Figure 27.24.](#)

**Figure 27.24. `tcp_debug`**

```
35 struct tcp_debug {  
36     n_time    td_time;           /* iptime(): ms since midnight, UTC */  
37     short     td_act;          /* TA_xxx value (Figure 27.25) */  
38     short     td_ostate;        /* old state */  
39     caddr_t   td_tcb;          /* addr of TCP connection block */  
40     struct tciphdr td_ti;      /* IP and TCP headers */  
41     short     td_req;          /* PRU_xxx value for TA_USER */  
42     struct tcpcb td_cb;        /* TCP connection block */  
43 };  
53 #define TCP_NDEBUG 100  
54 struct tcp_debug tcp_debug[TCP_NDEBUG];  
55 int     tcp_debx;
```

35-43

This is a large structure (196 bytes), since it contains a `tciphdr` structure with the IP and TCP headers and a TCP control block. Since the entire TCP control block can be printed by `trpt`. Also, if you are interested in, we can modify the source code (in `tcp.c`, release) to print whatever information we would like. The RTT variables in [Figure 25.28](#) were obtained using this code.

53-55

We also show the declaration of the array `tcp_debug` buffer. The index into the array (`tcp_debx`) is irrelevant.

almost 20,000 bytes.

There are only four calls to `tcp_trace` in the kernel value in the `td_act` member of the structure, as

### Figure 27.25. `td_act` values and corre

<code>td_act</code>	Description	Reference
<code>TA_DROP</code>	from <code>tcp_input</code> , when input segment is dropped after input processing complete, before call to <code>tcp_output</code>	Figure 29.27
<code>TA_INPUT</code>	after input processing complete, before call to <code>tcp_output</code>	Figure 29.26
<code>TA_OUTPUT</code>	before calling <code>ip_output</code> to send segment	Figure 26.32
<code>TA_USER</code>	from <code>tcp_usrreq</code> , after processing PRU_xxx request	Figure 30.1

Figure 27.27 shows the main body of the `tcp_trace` function that outputs directly to the console.

48-133

`ostate` is the old state of the connection, when this value and the new state of the connection can see the state transition that occurred. In Figure 27.27, we change the state of the connection, but the other fields remain the same.

### Sample Output

Figure 27.26 shows the first four lines of `tcpdump` output during a three-way handshake and the first data segment. (See Figure 25.12. (Appendix A of Volume 1 provides additional information about the `tcpdump` command and its output format.)

## Figure 27.26. tcpdump output from

```
1 0.0          bsdi.1025 > vangogh.discard: S 20288001:20288001(0)
2 0.362719 (0.3627)  vangogh.discard > bsdi.1025: S 3202722817:3202722817(0)
3 0.364316 (0.0016)  bsdi.1025 > vangogh.discard: . ack 1 win 4096
4 0.415859 (0.0515)  bsdi.1025 > vangogh.discard: . 1:513(512) ack 1 win 4096
```

Figure 27.28 shows the corresponding output fi

This output contains a few changes from the r decimal sequence numbers are printed as uns prints them as signed numbers). Some values have been output in decimal. The values from added to trpt by the authors, for Figure 25.28

## Figure 27.27. tcp\_trace function: save infor

```
tcp_debug.c
48 void
49 tcp_trace(act, ostate, tp, ti, req)
50 short act, ostate;
51 struct tcpcb *tp;
52 struct tcphdr *ti;
53 int req;
54 {
55     tcp_seq seq, ack;
56     int len, flags;
57     struct tcp_debug *td = &tcp_debug[tcp_debx++];
58     if (tcp_debx == TCP_NDEBUG)
59         tcp_debx = 0; /* circle back to start */
60     td->td_time = iptime();
61     td->td_act = act;
62     td->td_ostate = ostate;
63     td->td_tcb = (caddr_t) tp;
64     if (tp)
65         td->td_cb = *tp; /* structure assignment */
66     else
67         bzero((caddr_t) & td->td_cb, sizeof(*tp));
68     if (ti)
69         td->td_ti = *ti; /* structure assignment */
70     else
71         bzero((caddr_t) & td->td_ti, sizeof(*ti));
72     td->td_req = req;
73 #ifdef TCPDEBUG
74     if (tcpconsdebug == 0)
75         return;
76
77     /* output information on console */
78 }
79
132 #endif
133 }
```

tcp\_debug.c

**Figure 27.28. trpt output from ex**

```

953738 SYN_SENT: output 20288001:20288005(4) @0 (win=4096)
  <SYN> -> SYN_SENT
  rcv_nxt 0, rcv_wnd 0
  snd_una 20288001, snd_nxt 20288002, snd_max 20288002
  snd_wl1 0, snd_wl2 0, snd_wnd 0
  REXMT=12 (t_rxtshift=0), KEEP=150
  t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

953739 CLOSED: user CONNECT -> SYN_SENT
  rcv_nxt 0, rcv_wnd 0
  snd_una 20288001, snd_nxt 20288002, snd_max 20288002
  snd_wl1 0, snd_wl2 0, snd_wnd 0
  REXMT=12 (t_rxtshift=0), KEEP=150
  t_rtt=1, t_srtt=0, t_rttvar=24, t_rxtcur=12

954103 SYN_SENT: input 3202722817:3202722817(0) @20288002 (win=8192)
  <SYN,ACK> -> ESTABLISHED
  rcv_nxt 3202722818, rcv_wnd 4096
  snd_una 20288002, snd_nxt 20288002, snd_max 20288002
  snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
  KEEP=14400
  t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954103 ESTABLISHED: output 20288002:20288002(0) @3202722818 (win=4096)
  <ACK> -> ESTABLISHED
  rcv_nxt 3202722818, rcv_wnd 4096
  snd_una 20288002, snd_nxt 20288002, snd_max 20288002
  snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
  KEEP=14400
  t_rtt=0, t_srtt=16, t_rttvar=4, t_rxtcur=6

954153 ESTABLISHED: output 20288002:20288514(512) @3202722818 (win=4096)
  <ACK> -> ESTABLISHED
  rcv_nxt 3202722818, rcv_wnd 4096
  snd_una 20288002, snd_nxt 20288514, snd_max 20288514
  snd_wl1 3202722818, snd_wl2 20288002, snd_wnd 8192
  REXMT=6 (t_rxtshift=0), KEEP=14400
  t_rtt=1, t_srtt=16, t_rttvar=4, t_rxtcur=6

```

At time 953738 the SYN is sent. Notice that on millisecond time are outputit would take 8 digit midnight. The ending sequence number that is bytes are sent with the SYN, but these are the retransmit timer is 6 seconds (REXMT) and the (KEEP). These timer values are in 500-ms ticks segment is being timed for an RTT measuremei

This SYN segment is sent in response to the pre millisecond later the trace record for this system buffer. Even though the call to connect generat

to `tcp_trace` appears after processing the `PRU_RECV` records appear backward in the buffer. Also, while the connection state was `CLOSED`, and it changes to `SYN_RECV` from the first trace record to this one.

The third trace record, at time 954103, occurs shows a 362.7 ms difference.) This is how the "RTT (ms)" in [Figure 25.28](#) were computed. The connection state changes from `SYN_SENT` to `ESTABLISHED` when the segment is acknowledged. The RTT estimators are updated because the segment has been acknowledged.

The fourth trace record is the third segment of the other end's SYN. Since this segment contains all zeros (0).

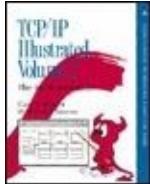
After the ACK has been sent at time 954103, the TCP process, which then calls `write` to send data, appears in trace record 5 at time 954153, 50 ms after the previous ACK is acknowledged. 512 bytes of data are sent, starting at offset 0. The retransmission timer is set to 3 seconds and the connection state is still `SYN_RECV`.

This output is caused by an application `write`. After the first four trace records, the next four are from `PRU_SENDBUF`. A `PRU_SENDBUF` request generates the output of the first 512 bytes of the send buffer. The other three do not cause output, since the connection state is still `SYN_RECV` during slow start. Four trace records are generated because the application example uses a TCP send buffer of 4096 and a receive buffer of 131072.

buffer is full, the process is put to sleep.

---

Team-Fly



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 27. TCP Functions

### 27.11 Summary

This chapter has covered a wide range of TCP functions that we'll encounter in the following chapters.

TCP connections can be aborted by sending an RST or they can be closed down gracefully, by sending a FIN and waiting for the four-way exchange of segments to complete.

Eight variables are stored in each routing table entry, three of which are updated when a connection is closed and six of which can be used later when a new connection is established. This lets the kernel keep track of certain variables, such

as the RTT estimators and the slow start threshold, between successive connections to the same destination. The system administrator can also set and lock some of these variables, such as the MTU, receive pipe size, and send pipe size, that affect TCP connections to that destination.

TCP is tolerant of received ICMP errors; none cause Net/3 to terminate an established connection. This handling of ICMP errors by Net/3 differs from earlier Berkeley releases.

Received TCP segments can arrive out of order and can contain duplicate data, and TCP must handle these anomalies. We saw that a reassembly queue is maintained for each connection, and this holds the out-of-order segments along with segments that arrive before they can be passed to the application.

Finally we looked at the type of information saved by the kernel when the SO\_DEBUG socket option is enabled for a TCP socket. This trace information can be a useful diagnostic tool in addition to

programs such as tcpdump.

## Exercises

**27.1** Why is the errno value 0 for the last row in [Figure 27.1](#)?

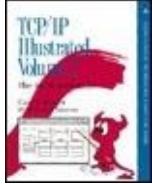
**27.2** What is the maximum value that can be stored in rmx\_rtt?

To save the route information in [Figure 27.3](#) for a given host, we enter a route into the routing table by hand for this destination. We then run the FTP client to send data to

**27.3** this host, making certain we send enough data, as described with [Figure 27.4](#). But after terminating the FTP client we look at the routing table, and all the values for this host are still 0. What's happening?

**Team-Fly**

Top



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 28. TCP Input

Section 28.1. Introduction

Section 28.2. Preliminary Processing

Section 28.3. `tcp_dooptions` Function

Section 28.4. Header Prediction

Section 28.5. TCP Input: Slow Path Processing

Section 28.6. Initiation of Passive Open, Completion of Active Open

Section 28.7. PAWS: Protection Against Wrapped Sequence Numbers

Section 28.8. Trim Segment so Data is Within Window

Section 28.9. Self-Connects and Simultaneous Opens

Section 28.10. Record Timestamp

## Section 28.11. RST Processing

## Section 28.12. Summary

---

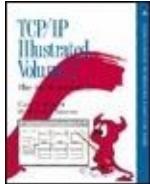
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.1 Introduction

TCP input processing is the largest piece of code that we examine in this text. The function `tcp_input` is about 1100 lines of code. The processing of incoming segments is not complicated, just long and detailed. Many implementations, including the one in Net/3, closely follow the input event processing steps in RFC 793, which spell out in detail how to respond to the various input segments, based on the current state of the connection.

The `tcp_input` function is called by `ipintr` (through the `pr_input` function in the protocol switch table) when a datagram is received with a protocol field of TCP.

`tcp_input` executes at the software interrupt level.

The function is so long that we divide its discussion into two chapters. [Figure 28.1](#) outlines the processing steps in `tcp_input`. This chapter discusses the steps through RST processing, and the next chapter starts with ACK processing.

## **Figure 28.1. Summary of TCP input processing steps.**

```

void
tcp_input()
{
    checksum TCP header and data;
findpcb:
    locate PCB for segment;
    if (not found)
        . goto dropwithreset;
    reset idle time to 0 and keepalive timer to 2 hours;
    process options if not LISTEN state;
    if (packet matched by header prediction) {
        completely process received segment;
        return;
    }

    switch (tp->t_state) {
    case TCPS_LISTEN:
        if SYN flag set, accept new connection request;
        goto trimthenstep6;

    case TCPS_SYN_SENT:
        if ACK of our SYN, connection completed;
trimthenstep6:
        trim any data not within window;
        goto step6;
    }

    process RFC 1323 timestamp;
    check if some data bytes are within the receive window;
    trim data segment to fit within window;

    if (RST flag set) {
        process depending on state;
        goto drop;
    }                                /* Chapter 28 finishes here */

    if (ACK flag set) {              /* Chapter 29 starts here */
        if (SYN_RCVD state)
            passive open or simultaneous open complete;
        if (duplicate ACK)
            fast recovery algorithm;
        update RTT estimators if segment timed;
        open congestion window;
        remove ACKed data from send buffer;
        change state if in FIN_WAIT_1, CLOSING, or LAST_ACK state;
    }

step6:
    update window information;
    process URG flag;

```

```

dodata:
    process data in segment, add to reassembly queue;
    if (FIN flag is set)
        process depending on state;
    if (SO_DEBUG socket option)
        tcp_trace(TA_INPUT);
    if (need output || ACK now)
        tcp_output();
    return;

dropafterack:
    tcp_output() to generate ACK;
    return;

dropwithreset:
    tcp_respond() to generate RST;
    return;

drop:
    if (SO_DEBUG socket option)
        tcp_trace(TA_DROP);
    return;
}

```

---

The first few steps are typical: validate the input segment (checksum, length, etc.) and locate the PCB for this connection. Given the length of the remainder of the function, however, an attempt is made to bypass all this logic with an algorithm called *header prediction* ([Section 28.4](#)). This algorithm is based on the assumption that segments are not typically lost or reordered, hence for a given connection TCP can often guess what the next received segment will be. If the header prediction algorithm works, notice that the function returns. This is the fast path through `tcp_input`.

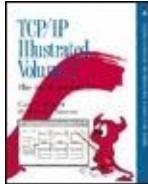
The slow path through the function ends

up at the label `dodata`, which tests a few flags and calls `tcp_output` if a segment should be sent in response to the received segment.

There are also three labels at the end of the function that are jumped to when errors occur: `dropafterack`, `dropwithreset`, and `drop`. The term *drop* means to drop the segment being processed, not drop the connection, but when an RST is sent by `dropwithreset` it normally causes the connection to be dropped.

The only other branching in the function occurs when a valid SYN is received in either the LISTEN or SYN\_SENT states, at the switch following header prediction. When the code at `trimthenstep6` finishes, it jumps to `step6`, which continues the normal flow.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

## 28.2 Preliminary Processing

Figure 28.2 shows the declarations and the initial processing of the received TCP segment.

**Figure 28.2. `tcp_input` function:  
declarations and preliminary processing.**

---

```

170 void
171 tcp_input(m, iphlen)
172 struct mbuf *m;
173 int     iphlen;
174 {
175     struct tcphdr *ti;
176     struct inpcb *inp;
177     caddr_t optp = NULL;
178     int     optlen;
179     int     len, tlen, off;
180     struct tcpcb *tp = 0;
181     int     tiflags;
182     struct socket *so;
183     int     todrop, acked, ourfinisacked, needoutput = 0;
184     short   ostate;
185     struct in_addr laddr;
186     int     dropsocket = 0;
187     int     iss = 0;
188     u_long  tiwin, ts_val, ts_ecr;
189     int     ts_present = 0;

190     tcpstat.tcp_ps_rcvtotal++;
191     /*
192      * Get IP and TCP header together in first mbuf.
193      * Note: IP leaves IP header in first mbuf.
194      */
195     ti = mtod(m, struct tcphdr *);
196     if (iphlen > sizeof(struct ip))
197         ip_strioptions(m, (struct mbuf *) 0);
198     if (m->m_len < sizeof(struct tcphdr)) {
199         if ((m = m_pullup(m, sizeof(struct tcphdr))) == 0) {
200             tcpstat.tcp_ps_rcvshort++;
201             return;
202         }
203         ti = mtod(m, struct tcphdr *);
204     }

```

---

*tcp\_input.c*

## Get IP and TCP headers in first mbuf

170-204

The argument iphlen is the length of the IP header, including possible IP options. If the length is greater than 20 bytes, options are present, and ip\_strioptions discards the options. TCP ignores all IP options other than a source route, which is saved specially by IP ([Section 9.6](#)) and fetched

later by TCP in [Figure 28.7](#). If the number of bytes in the first mbuf in the chain is less than the size of the combined IP/TCP header (40 bytes), `m_pullup` moves the first 40 bytes into the first mbuf.

The next piece of code, shown in [Figure 28.3](#), verifies the TCP checksum and offset field.

### Figure 28.3. `tcp_input` function: verify TCP checksum and offset field.

```
205  /*
206   * Checksum extended TCP header and data.
207   */
208  tlen = ((struct ip *) ti)->ip_len;
209  len = sizeof(struct ip) + tlen;
210  ti->ti_next = ti->ti_prev = 0;
211  ti->ti_x1 = 0;
212  ti->ti_len = (u_short) tlen;
213  HTONS(ti->ti_len);
214  if (ti->ti_sum = in_cksum(m, len)) {
215      tcpstat.tcpstat_rcvbadsum++;
216      goto drop;
217  }
218  /*
219   * Check that TCP offset makes sense,
220   * pull out TCP options and adjust length.      XXX
221   */
222  off = ti->ti_off << 2;
223  if (off < sizeof(struct tcphdr) || off > tlen) {
224      tcpstat.tcpstat_rcvbadoff++;
225      goto drop;
226  }
227  tlen -= off;
228  ti->ti_len = tlen;
```

## Verify TCP checksum

205-217

tlen is the TCP length, the number of bytes following the IP header. Recall that IP has already subtracted the IP header length from ip\_len. The variable len is then set to the length of the IP datagram, the number of bytes to be checksummed, including the pseudo-header. The fields in the pseudo-header are set, as required for the checksum calculation, as shown in [Figure 23.19](#).

## Verify TCP offset field

218-228

The TCP offset field, ti\_off, is the number of 32-bit words in the TCP header, including any TCP options. It is multiplied by 4 (to become the byte offset of the first data byte in the TCP segment) and checked for sanity. It must be greater than or equal to the size of the standard TCP header (20) and less than or equal to the TCP length.

The byte offset of the first data byte is subtracted from the TCP length, leaving tlen with the number of bytes of data in the segment (possibly 0). This value is stored back into the TCP header, in the variable ti\_len, and will be used throughout the function.

Figure 28.4 shows the next part of processing: handling of certain TCP options.

## Figure 28.4. `tcp_input` function: handle certain TCP options.

```
tcp_input.c
229     if (off > sizeof(struct tcphdr)) {
230         if (m->m_len < sizeof(struct ip) + off) {
231             if ({m = m_pullup(m, sizeof(struct ip) + off)) == 0} {
232                 tcpstat.tcpss_rcvshort++;
233                 return;
234             }
235             ti = mtod(m, struct tciphdr *);
236         }
237         optlen = off - sizeof(struct tcphdr);
238         optp = mtod(m, caddr_t) + sizeof(struct tciphdr);
239         /*
240          * Do quick retrieval of timestamp options ("options
241          * prediction?"). If timestamp is the only option and it's
242          * formatted as recommended in RFC 1323 Appendix A, we
243          * quickly get the values now and not bother calling
244          * tcp_dooptions(), etc.
245          */
246         if ((optlen == TCPOLEN_TSTAMP_APPA ||
247              (optlen > TCPOLEN_TSTAMP_APPA &&
248               optp[TCPOLEN_TSTAMP_APPA] == TCPOPT_EOL)) &&
249             (*{u_long *} optp == htonl(TCPOPT_TSTAMP_HDR)) &&
250             (ti->ti_flags & TH_SYN) == 0) {
251             ts_present = 1;
252             ts_val = ntohl(*{u_long *} (optp + 4));
253             ts_ecr = ntohl(*{u_long *} (optp + 8));
254             optp = NULL;           /* we've parsed the options */
255         }
256     }
```

## Get headers plus option into first mbuf

230-236

If the byte offset of the first data byte is greater than 20, TCP options are present. `m_pullup` is called, if necessary, to place the standard IP header, standard TCP header, and any TCP options in the first mbuf in the chain. Since the maximum size of these three pieces is 80 bytes (20+20+40), they all fit into the first packet header mbuf on the chain.

Since the only way `m_pullup` can fail here is when fewer than 20 plus off bytes are in the IP datagram, and since the TCP checksum has already been verified, we expect this call to `m_pullup` never to fail. Unfortunately the counter `tcps_rcvshort` is also shared by the call to `m_pullup` in [Figure 28.2](#), so looking at the counter doesn't tell us which call failed. Nevertheless, [Figure 24.5](#) shows that after receiving almost 9 million TCP segments, this counter is 0.

## Process timestamp option quickly

237-255

optlen is the number of bytes of options, and optp is a pointer to the first option byte. If the following three conditions are all true, only the timestamp option is present and it is in the desired format:

- 1. (a) The TCP option length equals 12 (TCPOLEN\_TSTAMP\_APPA), or (b) the TCP option length is greater than 12 and optp[12] equals the end-of-option byte.**

- The first 4 bytes of options equals 0x0101080a (TCPOPT\_TSTAMP\_HDR, which we described in [Section 26.6](#)).
- The SYN flag is not set (i.e., this segment is for an established connection, hence if a timestamp option is present, we know both sides have agreed to use the option).

If all three conditions are true, ts\_present is set to 1 the two timestamp values are fetched and stored in ts\_val and ts\_ecr;

and optp is set to null, since all the options have been parsed. The benefit in recognizing the timestamp option this way is to avoid calling the general option processing function `tcp_dooptions` later in the code. The general option processing function is OK for the other options that appear only with the SYN segment that creates a connection (the MSS and window scale options), but when the timestamp option is being used, it will appear with almost every segment on an established connection, so the faster it can be recognized, the better.

The next piece of code, shown in [Figure 28.5](#), locates the Internet PCB for the segment.

**Figure 28.5. `tcp_input` function: locate Internet PCB for segment.**

```

257     tiflags = ti->ti_flags;
258     /*
259      * Convert TCP protocol specific fields to host format.
260      */
261     NTOHL(ti->ti_seq);
262     NTOHL(ti->ti_ack);
263     NTOHS(ti->ti_win);
264     NTOHS(ti->ti_urp);
265     /*
266      * Locate pcb for segment.
267      */
268     findpcb:
269     inp = tcp_last_inpcb;
270     if (inp->inp_lport != ti->ti_dport ||
271         inp->inp_fport != ti->ti_sport ||
272         inp->inp_faddr.s_addr != ti->ti_src.s_addr ||
273         inp->inp_laddr.s_addr != ti->ti_dst.s_addr) {
274         inp = in_pcblockup(&tcb, ti->ti_src, ti->ti_sport,
275                            ti->ti_dst, ti->ti_dport, INPLOOKUP_WILDCARD);
276         if (inp)
277             tcp_last_inpcb = inp;
278         ++tcpstat.tcps_pcbcachemiss;
279     }

```

tcp\_input.c

## Save input flags and convert fields to host byte order

257-264

The received flags (SYN, FIN, etc.) are saved in the local variable tiflags, since they are referenced throughout the code. Two 16-bit values and the two 32-bit values in the TCP header are converted from network byte order to host byte order. The two 16-bit port numbers are left in network byte order, since the port numbers in the Internet PCB are in that order.

## Locate Internet PCB

265-279

TCP maintains a one-behind cache (`tcp_last_inpcb`) containing the address of the PCB for the last received TCP segment. This is the same technique used by UDP. The comparison of the four elements in the socket pair is in the same order as done by `udp_input`. If the cache entry does not match, `in_pcblklookup` is called, and the cache is set to the new PCB entry.

TCP does not have the same problem that we encountered with UDP: wildcard entries in the cache causing a high miss rate. The only time a TCP socket has a wildcard entry is for a server listening for connection requests. Once a connection is made, all four entries in the socket pair contain nonwildcard values. In [Figure 24.5](#) we see a cache hit rate of almost 80%.

[Figure 28.6](#) shows the next piece of code.

**Figure 28.6. `tcp_input` function: check if**

## segment should be dropped.

```
280  /*
281   * If the state is CLOSED (i.e., TCB does not exist) then
282   * all data in the incoming segment is discarded.
283   * If the TCB exists but is in CLOSED state, it is embryonic,
284   * but should either do a listen or a connect soon.
285   */
286  if (inp == 0)
287      goto dropwithreset;
288  tp = intotcpb(inp);
289  if (tp == 0)
290      goto dropwithreset;
291  if (tp->t_state == TCPS_CLOSED)
292      goto drop;
293  /* Unscale the window into a 32-bit value. */
294  if ((tiflags & TH_SYN) == 0)
295      tiwin = ti->ti_win << tp->snd_scale;
296  else
297      tiwin = ti->ti_win;
```

tcp\_input.c

## Drop segment and generate RST

280-287

If the PCB was not found, the input segment is dropped and an RST is sent as a reply. This is how TCP handles SYNs that arrive for a server that doesn't exist, for example. Recall that UDP sends an ICMP port unreachable in this case.

288-290

If the PCB exists but a corresponding TCP control block does not exist, the socket is probably being closed (tcp\_close releases

the TCP control block first, and then releases the PCB), so the input segment is dropped and an RST is sent as a reply.

## Silently drop segment

291-292

If the TCP control block exists, but the connection state is CLOSED, the socket has been created and a local address and local port may have been assigned, but neither connect nor listen has been called. The segment is dropped but nothing is sent as a reply. This scenario can happen if a client catches a server between the server's call to bind and 1isten. By silently dropping the segment and not replying with an RST, the client's connection request should time out, causing the client to retransmit the SYN.

## Unscale advertised window

293-297

If window scaling is to take place for this

connection, both ends must specify their send scale factor using the window scale option when the connection is established. If the segment contains a SYN, the window scale factor has not been established yet, so tiwin is copied from the value in the TCP header. Otherwise the 16-bit value in the header is left shifted by the send scale factor into a 32-bit value.

The next piece of code, shown in [Figure 28.7](#), does some preliminary processing if the socket debug option is enabled or if the socket is listening for incoming connection requests.

### **Figure 28.7. `tcp_input` function: handle debug option and listening sockets.**

```

298     so = inp->inp_socket;
299     if (so->so_options & (SO_DEBUG | SO_ACCEPTCONN)) {
300         if (so->so_options & SO_DEBUG) {
301             ostate = tp->t_state;
302             tcp_saveti = *ti;
303         }
304         if (so->so_options & SO_ACCEPTCONN) {
305             so = sonewconn(so, 0);
306             if (so == 0)
307                 goto drop;
308             /*
309             * This is ugly, but ....
310             *
311             * Mark socket as temporary until we're
312             * committed to keeping it. The code at
313             * 'drop' and 'dropwithreset' check the
314             * flag dropsocket to see if the temporary
315             * socket created here should be discarded.
316             * We mark the socket as discardable until
317             * we're committed to it below in TCPS_LISTEN.
318             */
319             dropsocket++;
320             inp = (struct inpcb *) so->so_pcb;
321             inp->inp_laddr = ti->ti_dst;
322             inp->inp_lport = ti->ti_dport;
323 #if BSD>=43
324             inp->inp_options = ip_srcroute();
325 #endif
326             tp = intotcpb(inp);
327             tp->t_state = TCPS_LISTEN;
328             /* Compute proper scaling value from buffer space */
329             while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
330                   TCP_MAXWIN << tp->request_r_scale < so->so_rcv.sb_hiwat)
331                 tp->request_r_scale++;
332         }
333     }

```

tcp\_input.c

## Save connection state and IP/TCP headers if socket debug option enabled

*300-303*

If the SO\_DEBUG socket option is enabled the current connection state is saved (ostate) as well as the IP and TCP headers (tcp\_saveti). These become arguments to `tcp_trace` when it is called at the end of the function ([Figure 29.26](#)).

## Create new socket if segment arrives for listening socket

304-319

When a segment arrives for a listening socket (SO\_ACCEPTCONN is enabled by listen), a new socket is created by sonewconn. This issues the protocol's PRU\_ATTACH request ([Figure 30.2](#)), which allocates an Internet PCB and a TCP control block. But more processing is needed before TCP commits to accept the connection request (such as the fundamental question of whether the segment contains a SYN or not), so the flag dropsocket is set, to cause the code at the labels drop and dropwithreset to discard the new socket if an error is encountered. If the received segment is OK, dropsocket is set back to 0 in [Figure 28.17](#).

320-326

inp and tp point to the new socket that has been created. The local address and local port are copied from the destination

address and destination port of the IP and TCP headers. If the input datagram contained a source route, it was saved by save\_rte. TCP calls ip\_srcroute to fetch that source route, saving a pointer to the mbuf containing the source route option in inp\_options. This option is passed to ip\_output by tcp\_output, and the reverse route is used for datagrams sent on this connection.

327

The state of the new socket is set to LISTEN. If the received segment contains a SYN, the code in [Figure 28.16](#) completes the connection request.

## Compute window scale factor

328-331

The window scale factor that will be requested is calculated from the size of the receive buffer. 65535 (TCP\_MAXWIN) is left shifted until the result exceeds the size of the receive buffer, or until the maximum

window scale factor is encountered (14, TCP\_MAX\_WINSHIFT). Notice that the requested window scale factor is chosen based on the size of the listening socket's receive buffer. This means the process must set the SO\_RCVBUF socket option before listening for incoming connection requests or it inherits the default value in tcp\_recvspace.

The maximum scale factor is 14, and  $65535 \times 2^{14}$  is 1,073,725,440. This is far greater than the maximum size of the receive buffer (262,144 in Net/3), so the loop should always terminate with a scale factor much less than 14. See [Exercises 28.1](#) and [28.2](#).

[Figure 28.8](#) shows the next part of TCP input processing.

**Figure 28.8. `tcp_input` function: reset idle time and keepalive timer, process options.**

```
334  /*
335   * Segment received on connection.
336   * Reset idle time and keepalive timer.
337   */
338   tp->t_idle = 0;
339   tp->t_timer[TCPT_KEEP] = tcp_keepidle;
340  /*
341   * Process options if not in LISTEN state,
342   * else do it below (after getting remote address).
343   */
344   if (optp && tp->t_state != TCPS_LISTEN)
345     tcp_dooptions(tp, optp, optlen, ti,
346                   &ts_present, &ts_val, &ts_ecr);
```

tcp\_input.c

## Reset idle time and keepalive timer

334-339

t\_idle is set to 0 since a segment has been received on the connection. The keep-alive timer is also reset to 2 hours.

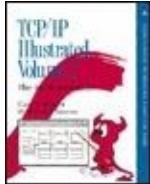
## Process TCP options if not in LISTEN state

340-346

If options are present in the TCP header, and if the connection state is not LISTEN, tcp\_dooptions processes the options. Recall that if only a timestamp option appears for an established connection, and that option is in the format recommended by [Appendix A](#) of RFC 1323, it was already

processed in [Figure 28.4](#) and optp was set to a null pointer. If the socket is in the LISTEN state, `tcp_dooptions` is called in [Figure 28.17](#) after the peer's address has been recorded in the PCB, because processing the MSS option requires knowledge of the route that will be used to this peer.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.3 `tcp_dooptions` Function

This function processes the five TCP options supported by Net/3 ([Section 26.4](#)): the EOL, NOP, MSS, window scale, and timestamp options. [Figure 28.9](#) shows the first part of this function.

**Figure 28.9. `tcp_dooptions` function:  
handle EOL and NOP options.**

```
1213 void  
1214 tcp_dooptions(tp, cp, cnt, ti, ts_present, ts_val, ts_ecr)  
1215 struct tcpcb *tp;  
1216 u_char *cp;  
1217 int cnt;  
1218 struct tciphdr *ti;  
1219 int *ts_present;  
1220 u_long *ts_val, *ts_ecr;  
1221 {  
1222     u_short mss;  
1223     int opt, optlen;  
1224     for (; cnt > 0; cnt -= optlen, cp += optlen) {  
1225         opt = cp[0];  
1226         if (opt == TCPOPT_EOL)  
1227             break;  
1228         if (opt == TCPOPT_NOP)  
1229             optlen = 1;  
1230         else {  
1231             optlen = cp[1];  
1232             if (optlen <= 0)  
1233                 break;  
1234         }  
1235         switch (opt) {  
1236             default:  
1237                 continue;  
1238         }  
1239     }  
1240 }
```

tcp\_input.c

## Fetch option type and length

1213-1229

The options are scanned and an EOL (end-of-options) terminates the processing, causing the function to return. The length of a NOP is set to 1, since this option is not followed by a length byte ([Figure 26.16](#)). The NOP will be ignored via the default in the switch statement.

1230-1234

All other options have a length byte that is

stored in optlen.

Any new options that are not understood by this implementation of TCP are also ignored. This occurs because:

- 1. Any new options defined in the future will have an option length (NOP and EOL are the only two without a length), and the for loop skips optlen bytes each time around the loop.**
  - The default in the switch statement ignores unknown options.

The final part of `tcp_dooptions`, shown in [Figure 28.10](#), handles the MSS, window scale, and timestamp options.

**Figure 28.10. `tcp_dooptions` function: process MSS, window scale, and timestamp options.**

```

1238     case TCPOPT_MAXSEG:
1239         if (optlen != TCPOLEN_MAXSEG)
1240             continue;
1241         if (!(ti->ti_flags & TH_SYN))
1242             continue;
1243         bcopy((char *) cp + 2, (char *) &mss, sizeof(mss));
1244         NTOHS(mss);
1245         (void) tcp_mss(tp, mss); /* sets t_maxseg */
1246         break;
1247
1248     case TCPOPT_WINDOW:
1249         if (optlen != TCPOLEN_WINDOW)
1250             continue;
1251         if (!(ti->ti_flags & TH_SYN))
1252             continue;
1253         tp->t_flags |= TF_RCVD_SCALE;
1254         tp->requested_s_scale = min(cp[2], TCP_MAX_WINSHIFT);
1255         break;
1256
1257     case TCPOPT_TIMESTAMP:
1258         if (optlen != TCPOLEN_TIMESTAMP)
1259             continue;
1260         *ts_present = 1;
1261         bcopy((char *) cp + 2, (char *) ts_val, sizeof(*ts_val));
1262         NTOHL(*ts_val);
1263         bcopy((char *) cp + 6, (char *) ts_ecr, sizeof(*ts_ecr));
1264         NTOHL(*ts_ecr);
1265
1266         /*
1267          * A timestamp received in a SYN makes
1268          * it ok to send timestamp requests and replies.
1269          */
1270         if (ti->ti_flags & TH_SYN) {
1271             tp->t_flags |= TF_RCVD_TSTMP;
1272             tp->ts_recent = *ts_val;
1273             tp->ts_recent_age = tcp_now;
1274         }
1275     }
1276 }
1277 }
```

tcp\_input.c

## MSS option

1238-1246

If the length is not 4 (TCPOLEN\_MAXSEG), or the segment does not have the SYN flag set, the option is ignored. Otherwise the 2 MSS bytes are copied into a local variable, converted to host byte order, and processed by tcp\_mss. This has the side

effect of setting the variable `t_maxseg` in the control block, the maximum number of bytes that can be sent in a segment to the other end.

## Window scale option

1247-1254

If the length is not 3 (TCPOLEN\_WINDOW), or the segment does not have the SYN flag set, the option is ignored. Net/3 remembers that it received a window scale request, and the scale factor is saved in `requested_s_scale`. Since only 1 byte is referenced by `cp[2]`, there can't be alignment problems. When the ESTABLISHED state is entered, if both ends requested window scaling, it is enabled.

## Timestamp option

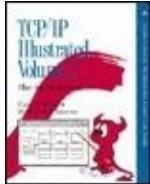
1255-1273

If the length is not 10 (TCPOLEN\_TIMESTAMP), the segment is

ignored. Otherwise the flag pointed to by ts\_present is set to 1, and the two timestamps are saved in the variables pointed to by ts\_val and ts\_ecr. If the received segment contains the SYN flag, Net/3 remembers that a timestamp request was received. ts\_recent is set to the received timestamp and ts\_recent\_age is set to tcp\_now, the counter of the number of 500-ms clock ticks since the system was initialized.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.4 Header Prediction

We now continue with the code in `tcp_input`, from where we left off in [Figure 28.8](#).

*Header prediction* was put into the 4.3BSD Reno release by Van Jacobson. The only description of the algorithm, other than the source code we're about to examine, is in [\[Jacobson 1990b\]](#), which is a copy of three slides showing the code.

Header prediction helps unidirectional data transfer by handling the two common cases.

- 1. If TCP is sending data, the next expected segment for this**

## **connection is an ACK for outstanding data.**

- If TCP is receiving data, the next expected segment for this connection is the next in-sequence data segment.

In both cases a small set of tests determines if the next expected segment has been received, and if so, it is handled in-line, faster than the general processing that follows later in this chapter and the next.

[[Partridge 1993](#)] shows an even faster version of TCP header prediction from a research implementation developed by Van Jacobson.

[Figure 28.11](#) shows the first part of header prediction.

**Figure 28.11. `tcp_input` function: header prediction, first part.**

---

```

347  /*
348   * Header prediction: check for the two common cases
349   * of a uni-directional data xfer. If the packet has
350   * no control flags, is in-sequence, the window didn't
351   * change and we're not retransmitting, it's a
352   * candidate. If the length is zero and the ack moved
353   * forward, we're the sender side of the xfer. Just
354   * free the data acked & wake any higher-level process
355   * that was blocked waiting for space. If the length
356   * is non-zero and the ack didn't move, we're the
357   * receiver side. If we're getting packets in order
358   * (the reassembly queue is empty), add the data to
359   * the socket buffer and note that we need a delayed ack.
360   */
361 if (tp->t_state == TCPS_ESTABLISHED &&
362     (tiflags & (TH_SYN | TH_FIN | TH_RST | TH_URG | TH_ACK)) == TH_ACK &&
363     (!ts_present || TSTMP_GEQ(ts_val, tp->ts_recent)) &&
364     ti->ti_seq == tp->recv_nxt &&
365     tiwin && tiwin == tp->snd_wnd &&
366     tp->snd_nxt == tp->snd_max) {
367
368     /*
369      * If last ACK falls within this segment's sequence numbers,
370      * record the timestamp.
371      */
372     if (ts_present && SEQ_LEQ(ti->ti_seq, tp->last_ack_sent) &&
373         SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len)) {
374       tp->ts_recent_age = tcp_now;
375       tp->ts_recent = ts_val;
376     }

```

---

tcp\_input.c

## Check if segment is the next expected

347-366

The following six conditions must *all* be true for the segment to be the next expected data segment or the next expected ACK:

### 1. The connection state must be ESTABLISHED.

- The following four control flags must not be on: SYN, FIN, RST, or URG. The ACK

flag must be on. In other words, of the six TCP control flags, the ACK flag must be set, the four just listed must be cleared, and it doesn't matter whether PSH is set or cleared. (Normally in the ESTABLISHED state the ACK flag is always on unless the RST flag is on.)

- If the segment contains a timestamp option, the timestamp value from the other end (`ts_val`) must be greater than or equal to the previous timestamp received for this connection (`ts_recent`). This is basically the PAWS test, which we describe in detail in [Section 28.7](#). If `ts_val` is less than `ts_recent`, this segment is out of order because it was sent before the most previous segment received on this connection. Since the other end always sends its timestamp clock (the global variable `tcp_now` in Net/3) as its timestamp value, the received timestamps of in-order segments always form a monotonic increasing sequence.

The timestamp need not increase with every in-order segment. Indeed, on a Net/3 system that increments the

timestamp clock (`tcp_now`) every 500 ms, multiple segments are often sent on a connection before that clock is incremented. Think of the timestamp and sequence number as forming a 64-bit value, with the sequence number in the low-order 32 bits and the timestamp in the high-order 32 bits. This 64-bit value always increases by at least 1 for every in-order segment (taking into account the modulo arithmetic).

- The starting sequence number of the segment (`ti_seq`) must equal the next expected receive sequence number (`rcv_nxt`). If this test is false, then the received segment is either a retransmission or a segment beyond the one expected.
- The window advertised by the segment (`tiwin`) must be nonzero, and must equal the current send window (`snd_wnd`). This means the window has not changed.
- The next sequence number to send (`snd_nxt`) must equal the highest sequence number sent (`snd_max`). This

means the last segment sent by TCP was not a retransmission.

## Update ts\_recent from received timestamp

367-375

If a timestamp option is present and if its value passes the test described with [Figure 26.18](#), the received timestamp (ts\_val) is saved in ts\_recent. Also, the current time (tcp\_now) is recorded in ts\_recent\_age.

Recall our discussion with [Figure 26.18](#) on how this test for a valid timestamp is flawed, and the correct test presented in [Figure 26.20](#). In this header prediction code the TSTMP\_GEQ test in [Figure 26.20](#) is redundant, since it was already done as step 3 of the if test at the beginning of [Figure 28.11](#).

The next part of the header prediction code, shown in [Figure 28.12](#), is for the sender of unidirectional data: process an ACK for outstanding data.

## Figure 28.12. tcp\_input function: header prediction, sender processing.

```
376     if (ti->ti_len == 0) {
377         if (SEQ_GT(ti->ti_ack, tp->snd_una) &&
378             SEQ_LEQ(ti->ti_ack, tp->snd_max) &&
379             tp->snd_cwnd >= tp->snd_wnd) {
380             /*
381             * this is a pure ack for outstanding data.
382             */
383             ++tcpstat.tcps_predack;
384             if (ts_present)
385                 tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
386             else if (tp->t_rtt &&
387                     SEQ_GT(ti->ti_ack, tp->t_rtseq))
388                 tcp_xmit_timer(tp, tp->t_rtt);
389
390             acked = ti->ti_ack - tp->snd_una;
391             tcpstat.tcps_rcvackpack++;
392             tcpstat.tcps_rcvackbyte += acked;
393             sbdrop(&so->so_snd, acked);
394             tp->snd_una = ti->ti_ack;
395             m_freem(m);
396
397             /*
398             * If all outstanding data is acked, stop
399             * retransmit timer, otherwise restart timer
400             * using current (possibly backed-off) value.
401             * If process is waiting for space,
402             * wakeup/selwakeup/signal. If data
403             * is ready to send, let tcp_output
404             * decide between more output or persist.
405             */
406             if (tp->snd_una == tp->snd_max)
407                 tp->t_timer[TCPT_REXMT] = 0;
408             else if (tp->t_timer[TCPT_PERSIST] == 0)
409                 tp->t_timer[TCPT_REXMT] = tp->t_rxtecur;
410
411             if (so->so_snd.sb_flags & SB_NOTIFY)
412                 sowakeup(so);
413             if (so->so_snd.sb_cc)
414                 (void) tcp_output(tp);
415             return;
416     }
417 }
```

tcp\_input.c

## Test for pure ACK

376-379

If the following four conditions are all true,  
this segment is a pure ACK.

## **1. The segment contains no data (ti\_len is 0).**

- The acknowledgment field in the segment (ti\_ack) is greater than the largest unacknowledged sequence number (snd\_una). Since this test is "greater than" and not "greater than or equal to," it is true only if some positive amount of data is acknowledged by the ACK.
- The acknowledgment field in the segment (ti\_ack) is less than or equal to the maximum sequence number sent (snd\_max).
- The congestion window (snd\_cwnd) is greater than or equal to the current send window (snd\_wnd). This test is true only if the window is fully open, that is, the connection is not in the middle of slow start or congestion avoidance.

## **Update RTT estimators**

384-388

If the segment contains a timestamp

option, or if a segment was being timed and the acknowledgment field is greater than the starting sequence number being timed, `tcp_xmit_timer` updates the RTT estimators.

## **Delete acknowledged bytes from send buffer**

389-394

`acked` is the number of bytes acknowledged by the segment. `sbdrop` deletes those bytes from the send buffer. The largest unacknowledged sequence number (`snd_una`) is set to the acknowledgment field and the received mbuf chain is released. (Since the length is 0, there should be just a single mbuf containing the headers.)

## **Stop retransmit timer**

395-407

If the received segment acknowledges all outstanding data (`snd_una` equals

`snd_max`), the retransmission timer is turned off. Otherwise, if the persist timer is off, the retransmit timer is restarted using `t_rxtcur` as the timeout.

Recall that when `tcp_output` sends a segment, it sets the retransmit timer only if the timer is not currently enabled. If two segments are sent one right after the other, the timer is set when the first is sent, but not touched when the second is sent. But if an ACK is received only for the first segment, the retransmit timer must be restarted, in case the second was lost.

## Awaken waiting processes

### 408-409

If a process must be awakened when the send buffer is modified, `sowakeup` is called. From [Figure 16.5](#), `SB_NOTIFY` is true if a process is waiting for space in the buffer, if a process is selecting on the buffer, or if a process wants the SIGIO signal for this socket.

## Generate more output

410-411

If there is data in the send buffer, `tcp_output` is called because the sender's window has moved to the right. `snd_una` was just incremented and `snd_wnd` did not change, so in [Figure 24.17](#) the entire window has shifted to the right.

The next part of header prediction, shown in [Figure 28.13](#), is the receiver processing when the segment is the next in-sequence data segment.

**Figure 28.13. `tcp_input` function: header prediction, receiver processing.**

```

414     } else if (ti->ti_ack == tp->snd_una &&
415                 tp->seg_next == (struct tciphdr *) tp &&
416                 ti->ti_len <= sbspace(&so->so_rcv)) {
417         /*
418          * this is a pure, in-sequence data packet
419          * with nothing on the reassembly queue and
420          * we have enough buffer space to take it.
421          */
422         ++tcpstat.tcpstat_preddat;
423         tp->rcv_nxt += ti->ti_len;
424         tcpstat.tcpstat_rcvpack++;
425         tcpstat.tcpstat_rcvbyte += ti->ti_len;
426         /*
427          * Drop TCP, IP headers and TCP options then add data
428          * to socket buffer.
429          */
430         m->m_data += sizeof(struct tciphdr) + off - sizeof(struct tcphdr);
431         m->m_len -= sizeof(struct tciphdr) + off - sizeof(struct tcphdr);
432         sbappend(&so->so_rcv, m);
433         sorwakeup(so);
434         tp->t_flags |= TF_DELACK;
435         return;
436     }
437 }
```

*tcp\_input.c*

## Test for next in-sequence data segment

**414-416**

If the following four conditions are all true, this segment is the next expected data segment for the connection, and there is room in the socket buffer for the data.

- 1. The amount of data in the segment (ti\_len) is greater than 0. This is the else portion of the if at the beginning of Figure 28.12.**

- The acknowledgment field (ti\_ack) equals the largest unacknowledged sequence number. This means no data is

acknowledged by this segment.

- The reassembly list of out-of-order segments for the connection is empty (seg\_next equals tp).
- There is room in the receive buffer for the data in the segment.

## Complete processing of received data

423-435

The next expected receive sequence number (rcv\_nxt) is incremented by the number of bytes of data. The IP header, TCP header, and any TCP options are dropped from the mbuf, and the mbuf chain is appended to the socket's receive buffer. The receiving process is awakened by sorwakeup. Notice that this code avoids calling the TCP\_REASS macro, since the tests performed by that macro have already been performed by the header prediction tests. The delayed-ACK flag is set and the input processing is complete.

## Statistics

How useful is header prediction? A few simple unidirectional transfers were run across a LAN (between bsdi and svr4, in both directions) and across a WAN (between vangogh.cs.berkeley.edu and ftp.uu.net in both directions). The netstat output ([Figure 24.5](#)) shows the two header prediction counters.

On the LAN, with no packet loss but a few duplicate ACKs, header prediction worked between 97 and 100% of the time. Across the WAN, however, the header prediction percentages dropped slightly to between 83 and 99%.

Realize that header prediction works on a per-connection basis, regardless how much additional TCP traffic is being received by the host, while the PCB cache works on a per-host basis. Even though lots of TCP traffic can cause PCB cache misses, if packets are not lost on a given connection, header prediction still works on that connection.

---

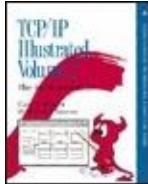
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.5 TCP Input: Slow Path Processing

We continue with the code that's executed if header prediction fails, the slow path through `tcp_input`. [Figure 28.14](#) shows the next piece of code, which prepares the received segment for input processing.

**Figure 28.14. `tcp_input` function: drop IP and TCP headers.**

---

```

438     /*
439      * Drop TCP, IP headers and TCP options.
440      */
441     m->m_data += sizeof(struct tcphdr) + off - sizeof(struct tcphdr);
442     m->m_len -= sizeof(struct tcphdr) + off - sizeof(struct tcphdr);

443     /*
444      * Calculate amount of space in receive window,
445      * and then do TCP input processing.
446      * Receive window is amount of space in rcv queue,
447      * but not less than advertised window.
448      */
449     {
450         int      win;

451         win = sbspace(&so->so_rcv);
452         if (win < 0)
453             win = 0;
454         tp->rcv_wnd = max(win, (int) (tp->rcv_adv - tp->rcv_nxt));
455     }

```

---

## Drop IP and TCP headers, including TCP options

438-442

The data pointer and length of the first mbuf in the chain are updated to skip over the IP header, TCP header, and any TCP options. Since off is the number of bytes in the TCP header, including options, the size of the normal TCP header (20) must be subtracted from the expression.

## Calculate receive window

443-455

win is set to the number of bytes available in the socket's receive buffer. rcv\_adv minus rcv\_nxt is the current advertised window. The receive window is the maximum of these two values. The max is taken to ensure that the value is not less than the currently advertised window. Also, if the process has taken data out of the socket receive buffer since the window was last advertised, win could exceed the advertised window, so TCP accepts up to win bytes of data (even though the other end should not be sending more than the advertised window).

This value is calculated now, since the code later in this function must determine how much of the received data (if any) fits within the advertised window. Any received data outside the advertised window is dropped: data to the left of the window is duplicate data that has already been received and acknowledged, and data to the right should not be sent by the other end.

Top

---

## Chapter 28. TCP Input

---

### 28.6 Initiation of Passive Open, Connection

If the state is LISTEN or SYN\_SENT, the code simply ignores the segment in these two states. If the segment in these two states is a SYN, and we'll drop it.

#### Initiation of Passive Open

Figure 28.15 shows the processing when the connection is initiated from the server's listening socket. The variables tp and inp refer to the new socket and the server's listening socket.

Figure 28.15. `tcp_input` function: checking for passive open

```

456     switch (tp->t_state) {
457         /*
458          * If the state is LISTEN then ignore segment if it contains an RST.
459          * If the segment contains an ACK then it is bad and send an RST.
460          * If it does not contain a SYN then it is not interesting; drop it.
461          * Don't bother responding if the destination was a broadcast.
462          * Otherwise initialize tp->rcv_nxt, and tp->irs, select an initial
463          * tp->iss, and send a segment:
464          *   <SEQ=ISS><ACK=RCV_NXT><CTL=SYN,ACK>
465          * Also initialize tp->snd_nxt to tp->iss+1 and tp->snd_una to tp->iss.
466          * Fill in remote peer address fields if not previously specified.
467          * Enter SYN RECEIVED state, and process any other fields of this
468          * segment in this state.
469          */
470     case TCPS_LISTEN: {
471         struct mbuf *am;
472         struct sockaddr_in *sin;
473         if (tiflags & TH_RST)
474             goto drop;
475         if (tiflags & TH_ACK)
476             goto dropwithreset;
477         if ((tiflags & TH_SYN) == 0)
478             goto drop;

```

tcp\_input.c

## Drop if RST, ACK, or no SYN

473-478

If the received segment contains the RST flag, dropped and an RST is sent as the reply. (The i few segments that does not contain an ACK.) It dropped. The remaining code for this case handles the LISTEN state. The new state will be SYN\_RECV.

Figure 28.16 shows the next piece of code for the TCP input function.

**Figure 28.16. tcp\_input function:**

```
479 /*  
480 * RFC1122 4.2.3.10, p. 104: discard bcast/mcast SYN  
481 * in_broadcast() should never return true on a received  
482 * packet with M_BCAST not set.  
483 */  
484 if (m->m_flags & (M_BCAST | M_MCAST) ||  
485     IN_MULTICAST(ti->ti_dst.s_addr))  
486     goto drop;  
  
487 am = m_get(M_DONTWAIT, MT_SONAME); /* XXX */  
488 if (am == NULL)  
489     goto drop;  
490 am->m_len = sizeof(struct sockaddr_in);  
491 sin = mtod(am, struct sockaddr_in *);  
492 sin->sin_family = AF_INET;  
493 sin->sin_len = sizeof(*sin);  
494 sin->sin_addr = ti->ti_src;  
495 sin->sin_port = ti->ti_sport;  
496 bzero((caddr_t) sin->sin_zero, sizeof(sin->sin_zero));  
  
497 laddr = inp->inp_laddr;  
498 if (inp->inp_laddr.s_addr == INADDR_ANY)  
499     inp->inp_laddr = ti->ti_dst;  
500 if (in_pcboptions(inp, am)) {  
501     inp->inp_laddr = laddr;  
502     (void) m_free(am);  
503     goto drop;  
504 }  
505 (void) m_free(am);
```

# Drop if broadcast or multicast

479-486

If the packet was sent to a broadcast or multicast address, it is destined for unicast applications. Recall that the M\_BCAST bit is set in the destination hardware address of the frame header based on the destination hardware address of the frame header and whether the IP address is a class D address.

The comment reference to `in_broadcast` is being multicasting) called that function here, to check if the broadcast address. The setting of the `M_BCAST` bit on the destination hardware address, was introduced in the previous section.

This Net/3 code tests only whether the destin

address, and does not call in\_broadcast to test for a broadcast address, on the assumption that a destination IP address that is a broadcast address is also a hardware broadcast address. This assumption is correct. Nevertheless, if a Net/3 system receives a SYN segment with a unicast hardware address, that segment will be discarded.

The destination address argument to IN\_MULTICAST is set in the order.

## Get mbuf for client's IP address and port

487-496

An mbuf is allocated to hold a sockaddr\_in structure containing the client's IP address and port number. The IP address header and the port number is copied from the structure. The structure is used shortly to connect the server's PCB to the client's PCB, and is then released.

The XXX comment is probably because of the code for the call to in\_pcbservice that follows. But Figure 24.5 shows that less than 2% of all received segments are broadcast.

## Set local address in PCB

497-499

laddr is the local address bound to the socket. In the normal scenario, the destination address is the address in the PCB. Note that the destination address is the interface of which local interface the datagram was received.

Notice that laddr cannot be the wildcard address 0.0.0.0, as it would lead to the destination IP address from the receive queue.

## Connect PCB to peer

500-505

in\_pcbconnect connects the server's PCB to the foreign process in the PCB. The mbuf is then released.

The next piece of code, shown in [Figure 28.17](#), is the complete implementation of the function.

**Figure 28.17. tcp\_input function: complete |**

```

506         tp->t_template = tcp_template(tp);
507         if (tp->t_template == 0) {
508             tp = tcp_drop(tp, ENOBUFS);
509             dropsocket = 0; /* socket is already gone */
510             goto drop;
511         }
512         if (optp)
513             tcp_dooptions(tp, optp, optlen, ti,
514                           &ts_present, &ts_val, &ts_ecr);
515         if (iss)
516             tp->iss = iss;
517         else
518             tp->iss = tcp_iss;
519         tcp_iss += TCP_ISSINCR / 2;
520         tp->iirs = ti->ti_seq;
521         tcp_sendseqinit(tp);
522         tcp_rcvseqinit(tp);
523         tp->t_flags |= TF_ACKNOW;
524         tp->t_state = TCPS_SYN_RECEIVED;
525         tp->t_timer[TCPTV_KEEP] = TCPTV_KEEP_INIT;
526         dropsocket = 0; /* committed to socket */
527         tcpstat.tcps_accepts++;
528         goto trimthenstep6;
529     }

```

*tcp\_input.c*

## Allocate and initialize IP and TCP header template

**506-511**

A template of the IP and TCP headers is created. [Figure 28.7](#) allocated the PCB and TCP control block header template.

## Process any TCP options

**512-514**

If TCP options are present, they are processed. [Figure 28.8](#) was done only if the connection was now for a listening socket, after the foreign address is used by the `tcp_mss` function: to get a route

or "foreign" (with regard to the peer's network

## Initialize ISS

515-519

The initial send sequence number is normally computed by incrementing by 64,000 (TCP\_ISSINCR divided by 4). However, its value is used instead of tcp\_iss to initialize the local iss variable.

The local iss variable is used for the following situations:

- A server is started on port 27 on the host with IP address 192.3.4.5. The socket pair on the client is (3000, 192.3.4.5). The socket pair on the server is (27, 192.3.4.5).
- A client on host 192.3.4.5 establishes a connection to a server on host 192.3.4.6. The socket pair on the client is (3000, 192.3.4.5). The socket pair on the server is (27, 192.3.4.6).
- The server actively closes the connection, putting it into the TIME\_WAIT state. While the connection is in this state, the local iss value is remembered in the TCP control block. Assuming the local iss value is 100,000.
- Before this connection leaves the TIME\_WAIT state, a new connection arrives from a client on host 192.3.4.5, port 3000. The sequence number of this new SYN is 200,000.
- Since this connection does not correspond to the connection in the TIME\_WAIT state, the local iss value is updated to 200,000.

code we just looked at is not executed. Instead, we'll see that it contains the following logic: if (200,000) is greater than the last sequence number then (1) the local variable iss is set to 100,000, (2) the TIME\_WAIT state is completely closed (its PCB is freed), and (3) a jump is made to findpcb ([Figure 28.5](#)).

- This time the server's listening PCB will be locked (not running), causing the code in this section to skip over the check. Instead, the value 228,000 is used in [Figure 28.17](#) to initialize the local variable iss.

This logic, which is allowed by RFC 1122, lets the server reuse the same socket pair as long as the server does the active connection. The variable tcp\_iss is incremented by 64,000 each time a new connection is opened (see [Section 30.4](#)): to ensure that if a single client reopens the connection repeatedly, a larger ISS is used each time, even if the client closes the connection, and even if the 500-ms timer (which controls the reuse of the last connection) has not yet expired.

## Initialize sequence number variables in control block

520-522

In [Figure 28.17](#), the initial receive sequence number is initialized to the value in the SYN segment. The following two macros are used to control the sequence number variables in the control block:

```
#define tcp_rcvseqinit(tp) \
    (tp)->rcv_adv = (tp)->rcv_nx + 1

#define tcp_sendseqinit(tp) \
    (tp)->snd_una = (tp)->snd_nx
    (tp)->iss
```

The addition of 1 in the first macro is because t

## ACK the SYN and change state

523-525

The TF\_ACKNOW flag is set since the ACK of a SYN becomes SYN\_RCVD, and the connection-establishment state (TCPTV\_KEEP\_INIT). Since the TF\_ACKNOW flag is set, the function `tcp_output` will be called. Looking at [Figure 24](#), we can see the segment with the SYN and ACK flags to be sent.

526-528

TCP is now committed to the new socket created. The code at `trimthenstep6` is jumped to. Remember that a SYN segment can contain data from the application until the connection enters the ESTABLISHED state.

## Completion of Active Open

Figure 28.18 shows the first part of processing TCP is expecting to receive a SYN.

**Figure 28.18. `tcp_input` function: check for SYN**

```
tcp_input.c
530     /*
531      * If the state is SYN_SENT:
532      * if seg contains an ACK, but not for our SYN, drop the input.
533      * if seg contains an RST, then drop the connection.
534      * if seg does not contain SYN, then drop it.
535      * Otherwise this is an acceptable SYN segment
536      * initialize tp->rcv_nxt and tp->irs
537      * if seg contains ack then advance tp->snd_una
538      * if SYN has been acked change to ESTABLISHED else SYN_RCVD state
539      * arrange for segment to be acked (eventually)
540      * continue processing rest of data/controls, beginning with URG
541      */
542 case TCPS_SYN_SENT:
543     if (((tiflags & TH_ACK) &&
544         (SEQ_LEQ(ti->ti_ack, tp->iss) ||
545          SEQ_GT(ti->ti_ack, tp->snd_max)))
546         goto dropwithreset;
547     if (tiflags & TH_RST) {
548         if (tiflags & TH_ACK)
549             tp = tcp_drop(tp, ECONNREFUSED);
550         goto drop;
551     }
552     if ((tiflags & TH_SYN) == 0)
553         goto drop;
```

tcp\_input.c

## Verify received ACK

530-546

When TCP sends a SYN in response to an active open, it ensures that the connection's iss is copied from the global variable `iss` (shown at the end of the previous section) is equal to the sequence number of the SYN segment.

shows the send sequence variables after the SY

## Figure 28.19. Send variables after SY

`tcp_sendseqinit` sets all four of these variables them to 366 when the SYN segment is output. [28.18](#) contains an ACK, and if the acknowledgement greater than `snd_max` (366), the ACK is invalid RST sent in reply. Notice that the received segment need not contain an ACK. It can contain only a ([Figure 24.15](#)), and is described shortly.

## Process and drop RST segment

547-551

If the received segment contains an RST, it is due to receipt of an acceptable ACK (which was expected). SYN is how the other end tells TCP that its connection request was accepted. This can be caused by the server process not being started or the socket's `so_error` variable, causing an error.

connect.

## Verify SYN flag set

552-553

If the SYN flag is not set in the received segme

The remainder of this case handles the receipt  
TCP's SYN. The next part of `tcp_input`, shown in

**Figure 28.20. `tcp_input` function: process r**

```
554     if (tiflags & TH_ACK) {                               tcp_input.c
555         tp->snd_una = ti->ti_ack;
556         if (SEQ_LT(tp->snd_nxt, tp->snd_una))
557             tp->snd_nxt = tp->snd_una;
558     }
559     tp->t_timer[TCPT_REXMT] = 0;
560     tp->irs = ti->ti_seq;
561     tcp_rcvseqinit(tp);
562     tp->t_flags |= TF_ACKNOW;
563     if (tiflags & TH_ACK && SEQ_GT(tp->snd_una, tp->iss)) {
564         tcpstat.tcps_connects++;
565         soisconnected(so);
566         tp->t_state = TCPS_ESTABLISHED;
567         /* Do window scaling on this connection? */
568         if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
569             (TF_RCVD_SCALE | TF_REQ_SCALE)) {
570             tp->snd_scale = tp->requested_s_scale;
571             tp->rcv_scale = tp->request_r_scale;
572         }
573         (void) tcp_reass(tp, (struct tciphdr *) 0,
574                           (struct mbuf *) 0);
575         /*
576          * if we didn't have to retransmit the SYN,
577          * use its rtt as our initial srtt & rtt var.
578          */
579         if (tp->t_rtt)
580             tcp_xmit_timer(tp, tp->t_rtt);
581     } else
582         tp->t_state = TCPS_SYN_RECEIVED;
```

## Process ACK

554-558

If the received segment contains an ACK, `snd_una` becomes 366, since 366 acknowledgment field. If `snd_nxt` is less than `s` 28.19), `snd_nxt` is set to `snd_una`.

## Turn off retransmission timer

559

The retransmission timer is turned off.

This is a bug. This timer should be turned off a SYN without an ACK is a simultaneous open TCP's SYN.

## Initialize receive sequence numbers

560-562

The initial receive sequence number is copied from the segment. The `tcp_rcvseqinit` macro (shown at the bottom) copies `rcv_adv` and `rcv_nxt` to the receive sequence number, causing `tcp_output` to be called at the bottom of the function.

contain `rcv_nxt` as the acknowledgment field (`F` received).

**563-564**

If the received segment contains an ACK, and i connection, the active open is complete, and th

This second test appears superfluous. At the l the received acknowledgment field if the ACK statement in [Figure 28.18](#) verified that the re the ISS. So at this point in the code, if the AC `snd_una` is greater than the ISS.

## **Connection is established**

**565-566**

`soisconnected` sets the socket state to connecte ESTABLISHED.

## **Check for window scale option**

**567-572**

If TCP sent the window scale option in its SYN : the option is enabled and the two variables `snd`

control block is initialized to 0 by `tcp_newtcpb`  
the window scale option is not used.

## Pass any queued data to process

573-574

Since data can arrive for a connection before it  
now placed in the receive buffer by calling `tcp_`  
argument.

This test is unnecessary. In this piece of code,  
that moves it from the `SYN_SENT` state to the  
this received SYN segment, it isn't processed  
function. If TCP just received a SYN without a  
data, that data is handled later ([Figure 29.2](#))  
connection from the `SYN_RCVD` state to the E

Although it is valid for data to accompany a S  
segment correctly, Net/3 never generates suc

## Update RTT estimators

575-580

If the SYN that is ACKed was being timed, `tcp_`  
on the measured RTT for the SYN.

TCP ignores a received timestamp option here  
a timestamp in a SYN generated by an active  
agrees to the option, the other end should echo  
echoes the received timestamp in a SYN in Figure 28.21.  
received timestamp here, instead of t\_rtt, but  
there's no advantage in using the timestamp  
timestamp option, instead of the t\_rtt counter  
are in flight at once, providing more RTT timer

## Simultaneous open

581-582

When TCP receives a SYN without an ACK in the stack  
and the connection moves to the SYN\_RCVD state.

The next piece of code, shown in Figure 28.21,  
label trimthenstep6 is also jumped to at the end of the function.

**Figure 28.21. `tcp_input` function: code for simultaneous open**

```

583     trimthenstep6:
584     /*
585      * Advance ti->ti_seq to correspond to first data byte.
586      * If data, trim to stay within window,
587      * dropping FIN if necessary.
588      */
589     ti->ti_seq++;
590     if (ti->ti_len > tp->rcv_wnd) {
591         todrop = ti->ti_len - tp->rcv_wnd;
592         m_adj(m, -todrop);
593         ti->ti_len = tp->rcv_wnd;
594         tiflags &= ~TH_FIN;
595         tcpstat.tcps_rcvpackafterwin++;
596         tcpstat.tcps_rcvbyteafterwin += todrop;
597     }
598     tp->snd_w1 = ti->ti_seq - 1;
599     tp->rcv_up = ti->ti_seq;
600     goto step6;
601 }
```

tcp\_input.c

## 584-589

The sequence number of the segment is incremented by any data in the segment, ti\_seq now contains the sequence number of the first byte of data.

## Drop any received data that follows receive window

## 590-597

ti\_len is the number of data bytes in the segment. If excess data (ti\_len minus rcv wnd) is dropped, the function causes the data to be trimmed from the segment and updated to be the new amount of data in the segment. The FIN flag is cleared. This is because the FIN would follow the data because it was outside the receive window.

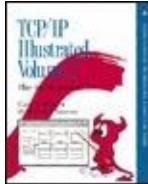
If too much data is received with a SYN, and if the other end received TCP's SYN, which contains

the other end ignored the advertised window  
much data accompanies a SYN performing an  
window advertisement, so it has to guess how

## Force update of window variables

598-599

snd\_wl1 is set the received sequence number r causes the three window update variables, snd. The receive urgent pointer (rcv\_up) is set to th to step6, which refers to a step in RFC 793, and



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.7 PAWS: Protection Against Wrapped Sequence Numbers

The next part of `tcp_input`, shown in [Figure 28.22](#), provides protection against wrapped sequence numbers: the PAWS algorithm from RFC 1323. Also recall our discussion of the timestamp option in [Section 26.6](#).

**Figure 28.22. `tcp_input` function: process timestamp option.**

---

```

602     /*
603      * States other than LISTEN or SYN_SENT.
604      * First check timestamp, if present.
605      * Then check that at least some bytes of segment are within
606      * receive window. If segment begins before rcv_nxt,
607      * drop leading data (and SYN); if nothing left, just ack.
608      *
609      * RFC 1323 PAWS: If we have a timestamp reply on this segment
610      * and it's less than ts_recent, drop it.
611      */
612     if (ts_present && (tiflags & TH_RST) == 0 && tp->ts_recent &&
613         TSTMP_LT(tp_val, tp->ts_recent)) {
614
615         /* Check to see if ts_recent is over 24 days old. */
616         if ((int) (tcp_now - tp->ts_recent_age) > TCP_PAWS_IDLE) {
617             /*
618              * Invalidate ts_recent. If this segment updates
619              * ts_recent, the age will be reset later and ts_recent
620              * will get a valid value. If it does not, setting
621              * ts_recent to zero will at least satisfy the
622              * requirement that zero be placed in the timestamp
623              * echo reply when ts_recent isn't valid. The
624              * age isn't reset until we get a valid ts_recent
625              * because we don't want out-of-order segments to be
626              * dropped when ts_recent is old.
627            */
628             tp->ts_recent = 0;
629         } else {
630             tcpstat.tcp_ps_rcvdupack++;
631             tcpstat.tcp_ps_rcvdupbyte += ti->ti_len;
632             tcpstat.tcp_ps_pawsdrop++;
633             goto dropafterack;
634         }

```

---

tcp\_input.c

## Basic PAWS test

602-613

ts\_present was set by tcp\_dooptions if a timestamp option was present. If the following three conditions are all true, the segment is dropped:

### 1. the RST flag is not set (**Exercise 28.8**),

- TCP has received a valid timestamp from

this peer (`ts_recent` is nonzero), and

- the received timestamp in this segment (`ts_val`) is less than the previously received timestamp from this peer.

PAWS is built on the premise that the 32-bit timestamp values wrap around at a much lower frequency than the 32-bit sequence numbers, on a high-speed connection. [Exercise 28.6](#) shows that even at the highest possible timestamp counter frequency (incrementing by 1 bit every millisecond), the sign bit of the timestamp wraps around only every 24 days. On a high-speed network such as a gigabit network, the sequence number can wrap in 17 seconds (Section 24.3 of Volume 1). Therefore, if the received timestamp value is less than the most recent one from this peer, this segment is old and must be discarded (subject to the outdated timestamp test that follows). The packet might be discarded later in the input processing because the sequence number is "old," but PAWS is intended for high-speed connections where the sequence numbers can wrap quickly.

Notice that the PAWS algorithm is symmetric: it not only discards duplicate data segments but also discards duplicate ACKs. All received segments are subject to PAWS. Recall that the header prediction code also applied the PAWS test ([Figure 28.11](#)).

## Check for outdated timestamp

614-627

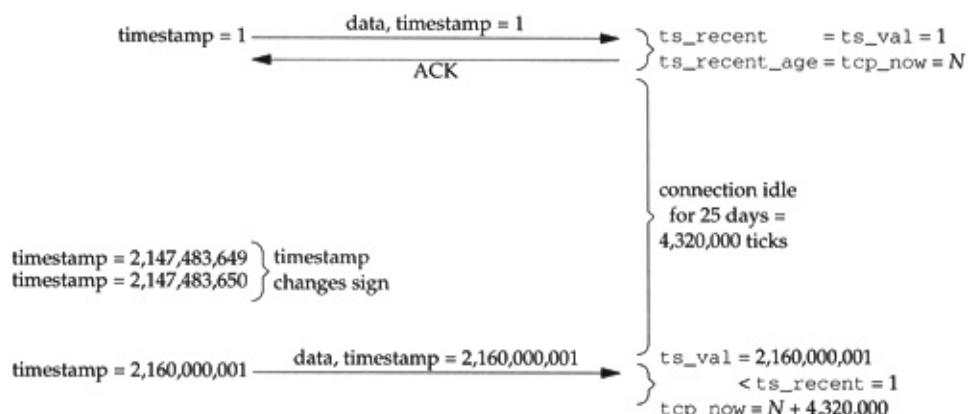
There is a small possibility that the reason the PAWS test fails is because the connection has been idle for a long time. The received segment is not a duplicate; it is just that because the connection has been idle for so long, the peer's timestamp value has wrapped around when compared to the most recent timestamp from that peer.

Whenever `ts_recent` is copied from the timestamp in a received segment, `ts_recent_age` records the current time (`tcp_now`). If the time at which `ts_recent` was saved is more than 24 days ago, it is

set to 0 to invalidate it. The constant `TCP_PAWS_IDLE` is defined to be  $(24 \times 24 \times 60 \times 60 \times 2)$ , the final 2 being the number of ticks per second. The received segment is not dropped in this case, since the problem is not a duplicated segment, but an outdated timestamp. See also [Exercises 28.6](#) and [28.7](#).

[Figure 28.23](#) shows an example of an outdated timestamp. The system on the left is a non-Net/3 system that increments its timestamp clock at the highest frequency allowed by RFC 1323: once every millisecond. The system on the right is a Net/3 system.

### Figure 28.23. Example of outdated timestamp.

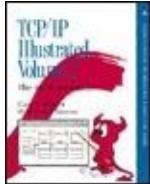


When the data segment arrives with a timestamp of 1, that value is saved in `ts_recent` and `ts_recent_age` is set to the current time (`tcp_now`), as shown in [Figures 28.11](#) and [28.35](#). The connection is then idle for 25 days, during which time `tcp_now` will increase by 4,320,000 ( $25 \times 24 \times 60 \times 60 \times 2$ ). During these 25 days the other end's timestamp clock will increase by 2,160,000,000 ( $25 \times 24 \times 60 \times 60 \times 1000$ ). During this interval the timestamp "changes sign" with regard to the value 1, that is, 2,147,483,649 is greater than 1, but 2,147,483,650 is less than 1 (recall [Figure 24.26](#)). Therefore, when the data segment is received with a timestamp of 2,160,000,001, this value is less than `ts_recent` (1), when compared using the `TSTMP_LT` macro, so the PAWS test fails. But since `tcp_now` minus `ts_recent_age` is greater than 24 days, the reason for the failure is that the connection has been idle for more than 24 days, and the segment is accepted.

## Drop duplicate segment

The segment is determined to be a duplicate based on the PAWS algorithm, and the timestamp is not outdated. It is dropped, after being acknowledged (since all duplicate segments are acknowledged).

[Figure 24.5](#) shows a much smaller value for tcps\_pawsdrop (22) than for tcps\_rcvduppck (46,953). This is probably because fewer systems support the timestamp option today, causing most duplicate packets to be discarded by later tests in TCP's input processing instead of by PAWS.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.8 Trim Segment so Data is Within Window

This section trims the received segment so that it contains only data that is within the advertised window:

- duplicate data at the beginning of the received segment is discarded, and
- data that is beyond the end of the window is discarded from the end of the segment.

What remains is new data within the window. The code shown in [Figure 28.24](#) checks if there is any duplicate data at the beginning of the segment.

## Figure 28.24. `tcp_input` function: check for duplicate data at beginning of segment.

```
635     todrop = tp->recv_nxt - ti->ti_seq;           —tcp_input.c
636     if (todrop > 0) {
637         if (tiflags & TH_SYN) {
638             tiflags &= ~TH_SYN;
639             ti->ti_seq++;
640             if (ti->ti_urp > 1)
641                 ti->ti_urp--;
642             else
643                 tiflags &= ~TH_URG;
644             todrop--;
645     }           —tcp_input.c
```

### Check if any duplicate data at front of segment

635-636

If the starting sequence number of the received segment (`ti_seq`) is less than the next receive sequence number expected (`recv_nxt`), data at the beginning of the segment is old and `todrop` will be greater than 0. These data bytes have already been acknowledged and passed to the application (Figure 24.18).

### Remove duplicate SYN

637-645

If the SYN flag is set, it refers to the first sequence number in the segment, which is known to be old. The SYN flag is cleared and the starting sequence number of the segment is incremented by 1 to skip over the duplicate SYN. Furthermore, if the urgent offset in the received segment ( $ti\_urp$ ) is greater than 1, it must be decremented by 1, since the urgent offset is relative to the starting sequence number, which was just incremented. If the urgent offset is 0 or 1, it is left alone, but in case it was 1, the URG flag is cleared. Finally `todrop` is decremented by 1 (since the SYN occupies a sequence number).

The handling of duplicate data at the front of the segment continues in [Figure 28.25](#).

**Figure 28.25. `tcp_input` function: handle completely duplicate segment.**

```

646     if (todrop >= ti->ti_len) {
647         tcpstat.tcpst_rcvdupack++;
648         tcpstat.tcpst_rcvdupbyte += ti->ti_len;
649         /*
650          * If segment is just one to the left of the window,
651          * check two special cases:
652          * 1. Don't toss RST in response to 4.2-style keepalive.
653          * 2. If the only thing to drop is a FIN, we can drop
654          *    it, but check the ACK or we will get into FIN
655          *    wars if our FINs crossed (both CLOSING).
656          * In either case, send ACK to resynchronize,
657          * but keep on processing for RST or ACK.
658          */
659         if ((tiflags & TH_FIN && todrop == ti->ti_len + 1)
660             )
661             todrop = ti->ti_len;
662             tiflags &= ~TH_FIN;
663             tp->t_flags |= TF_ACKNOW;
664         } else {
665             /*
666              * Handle the case when a bound socket connects
667              * to itself. Allow packets with a SYN and
668              * an ACK to continue with the processing.
669              */
670             if (todrop != 0 || (tiflags & TH_ACK) == 0)
671                 goto dropafterack;
672             }
673         } else {
674             tcpstat.tcpst_rcvpardupack++;
675             tcpstat.tcpst_rcvpardupbyte += todrop;
676         }
677         m_adj(m, todrop);
678         ti->ti_seq += todrop;
679         ti->ti_len -= todrop;
680         if (ti->ti_urp > todrop)
681             ti->ti_urp -= todrop;
682         else {
683             tiflags &= ~TH_URG;
684             ti->ti_urp = 0;
685         }
686     }

```

---

tcp\_input.c

## Check for entire duplicate packet

646-648

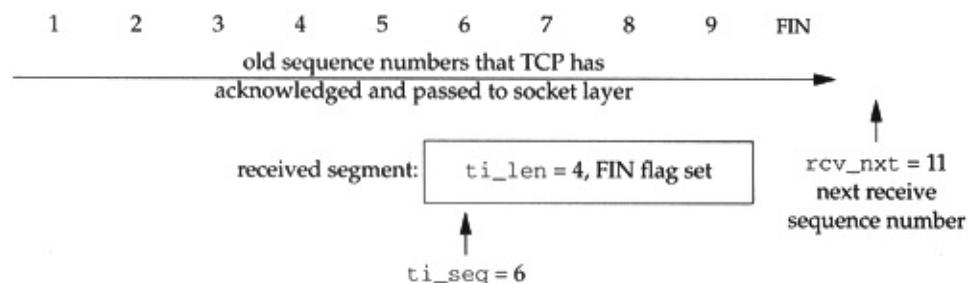
If the amount of duplicate data at the front of the segment is greater than or equal to the size of the segment, the entire segment is a duplicate.

## Check for duplicate FIN

649-663

The next check is whether the FIN is duplicated. [Figure 28.26](#) shows an example of this.

**Figure 28.26. Example of duplicate packet with FIN flag set.**



In this example  $todrop$  equals 5, which is greater than or equal to  $ti\_len$  (4). Since the FIN flag is set and  $todrop$  equals  $ti\_len$  plus 1,  $todrop$  is set to 4, the FIN flag is cleared, and the TF\_ACKNOW flag is set, forcing an immediate ACK to be sent at the end of this function. This example also works for other segments if  $ti\_seq$  plus  $ti\_len$  equals 10.

The code contains the comment regarding 4.2BSD keepalives. This code (another test within the if statement) is omitted.

## Generate duplicate ACK

664-672

If `todrop` is nonzero (the completely duplicate segment contains data) or the ACK flag is not set, the segment is dropped and an ACK is generated by `dropafterack`. This normally occurs when the other end did not receive our ACK, causing the other end to retransmit the segment. TCP generates another ACK.

## Handle simultaneous open or self-connect

664-672

This code also handles either a simultaneous open or a socket that connects to itself. We go over both of these scenarios in the next section. If

`todrop` equals 0 (there is no data in the completely duplicate segment) and the ACK flag is set, processing is allowed to continue.

This if statement is new with 4.4BSD. Earlier Berkeley-derived systems just had a jump to `dropafterack`. These systems could not handle either a simultaneous open or a socket connecting to itself.

Nevertheless, the piece of code in this figure still has bugs, which we describe at the end of this section.

## Update statistics for partial duplicate segments

673-676

This else clause is executed when `todrop` is less than the segment length: only part of the segment contains duplicate bytes.

## Remove duplicate data and update urgent offset

677-685

The duplicate bytes are removed from the front of the mbuf chain by m\_adj and the starting sequence number and length adjusted appropriately. If the urgent offset points to data still in the mbuf, it is also adjusted. Otherwise the urgent offset is set to 0 and the URG flag is cleared.

The next part of input processing, shown in [Figure 28.27](#), handles data that arrives after the process has terminated.

### Figure 28.27. `tcp_input` function: handle data that arrives after the process terminates.

```
687  /*
688   * If new data is received on a connection after the
689   * user processes are gone, then RST the other end.
690   */
691  if ((so->so_state & SS_NOFDREF) &&
692      tp->t_state > TCPS_CLOSE_WAIT && ti->ti_len) {
693      tp = tcp_close(tp);
694      tcpstat.tcps_rcvafterclose++;
695      goto dropwithreset;
696 }
```

687-696

If the socket has no descriptor referencing

it, the process has closed the connection (the state is any one of the five with a value greater than CLOSE\_WAIT in [Figure 24.16](#)), and there is data in the received segment, the connection is closed. The segment is then dropped and an RST is output.

Because of TCP's half-close, if a process terminates unexpectedly (perhaps it is terminated by a signal), when the kernel closes all open descriptors as part of process termination, a FIN is output by TCP. The connection moves into the FIN\_WAIT\_1 state. But the receipt of the FIN by the other end doesn't tell TCP whether this end performed a half-close or a full-close. If the other end assumes a half-close, and sends more data, it will receive an RST from the code in [Figure 28.27](#).

The next piece of code, shown in [Figure 28.29](#), removes any data from the end of the received segment that is beyond the right edge of the advertised window.

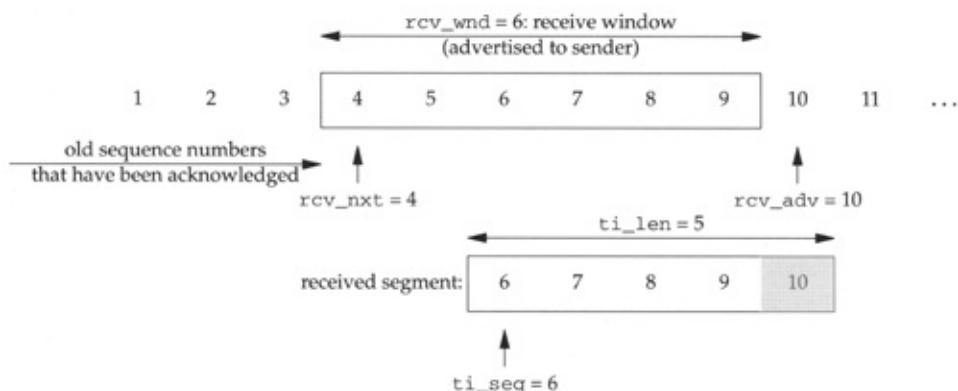
## Calculate number of bytes beyond right

## edge of window

697-703

todrop contains the number of bytes of data beyond the right edge of the window. For example, in [Figure 28.28](#), todrop would be (6+5) minus (4+6), or 1.

**Figure 28.28. Example of received segment with data beyond right edge of window.**



**Figure 28.29. `tcp_input` function: remove data beyond right edge of window.**

```

697  /*
698   * If segment ends after window, drop trailing data
699   * (and PUSH and FIN); if nothing left, just ACK.
700   */
701  todrop = (ti->ti_seq + ti->ti_len) - (tp->recv_nxt + tp->recv_wnd);
702  if (todrop > 0) {
703      tcpstat.tcpstat_rcvpackafterwin++;
704      if (todrop >= ti->ti_len) {
705          tcpstat.tcpstat_rcvbyteafterwin += ti->ti_len;
706          /*
707           * If a new connection request is received
708           * while in TIME_WAIT, drop the old connection
709           * and start over if the sequence numbers
710           * are above the previous ones.
711           */
712      if (tiflags & TH_SYN &&
713          tp->t_state == TCPS_TIME_WAIT &&
714          SEQ_GT(ti->ti_seq, tp->recv_nxt)) {
715          iss = tp->recv_nxt + TCP_ISSINCR;
716          tp = tcp_close(tp);
717          goto findpcb;
718      }
719      /*
720       * If window is closed can only take segments at
721       * window edge, and have to drop data and PUSH from
722       * incoming segments. Continue processing, but
723       * remember to ack. Otherwise, drop segment
724       * and ack.
725       */
726      if (tp->recv_wnd == 0 && ti->ti_seq == tp->recv_nxt) {
727          tp->t_flags |= TF_ACKNOW;
728          tcpstat.tcpstat_rcvwinprobe++;
729      } else
730          goto dropafterack;
731      } else
732          tcpstat.tcpstat_rcvbyteafterwin += todrop;
733      m_adj(m, -todrop);
734      ti->ti_len -= todrop;
735      tiflags &= ~(TH_PUSH | TH_FIN);
736  }

```

tcp\_input.c

## Check for new incarnation of a connection in the TIME\_WAIT state

**704-718**

If todrop is greater than or equal to the length of the segment, the entire segment will be dropped. If the following three conditions are all true:

## **1. the SYN flag is set, and**

- the connection is in the TIME\_WAIT state, and
- the new starting sequence number is greater than the final sequence number for the connection,

this is a request for a new incarnation of a connection that was recently terminated and is currently in the TIME\_WAIT state. This is allowed by RFC 1122, but the ISS for the new connection must be greater than the last sequence number used (rcv\_nxt). TCP adds 128,000 (TCP\_ISSINCR), which becomes the ISS when the code in [Figure 28.17](#) is executed. The PCB and TCP control block for the connection in the TIME\_WAIT state is discarded by tcp\_close. A jump is made to findpcb ([Figure 28.5](#)) to locate the PCB for the listening server, assuming it is still running. The code in [Figure 28.7](#) is then executed, creating a new socket for the new connection, and finally the code in [Figures 28.16](#) and [28.17](#) will complete the new connection request.

## **Check for probe of closed window**

719-728

If the receive window is closed (`rcv_wnd` equals 0) and the received segment starts at the left edge of the window (`rcv_nxt`), then the other end is probing TCP's closed window. An immediate ACK is sent as the reply, even though the ACK may still advertise a window of 0. Processing of the received segment also continues for this case.

## **Drop other segments that are completely outside window**

729-730

The entire segment lies outside the window and it is not a window probe, so the segment is discarded and an ACK is sent as the reply. This ACK will contain the expected sequence number.

## **Handle segments that contain some**

## valid data

731-735

The data to the right of the window is discarded from the mbuf chain by `m_adj` and `ti_len` is updated. In the case of a probe into a closed window, this discards all the data in the mbuf chain and sets `ti_len` to 0. Finally the FIN and PSH flags are cleared.

## When to Drop an ACK

The code in [Figure 28.25](#) has a bug that causes a jump to `dropafterack` in several cases when the code should fall through for further processing of the segment [[Carlson 1993](#); [Lanciani 1993](#)]. In an actual scenario, when both ends of a connection had a hole in the data on the reassembly queue and both ends enter the persist state, the connection becomes deadlocked as both ends throw away perfectly good ACKs.

The fix is to simplify the code at the

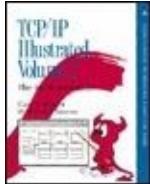
beginning of [Figure 28.25](#). Instead of jumping to dropafterack, a completely duplicate segment causes the FIN flag to be turned off and an immediate ACK to be generated at the end of the function. Lines 646676 in [Figure 28.25](#) are replaced with the code shown in [Figure 28.30](#). This code also corrects another bug present in the original code ([Exercise 28.9](#)).

## Figure 28.30. Correction for lines 646676 of Figure 28.25.

---

```
if (todrop > ti->ti_len ||  
    todrop == ti->ti_len && (tiflags & TH_FIN) == 0) {  
  
    /*  
     * Any valid FIN must be to the left of the window.  
     * At this point the FIN must be a duplicate or  
     * out of sequence; drop it.  
    */  
    tiflags &= ~TH_FIN;  
  
    /*  
     * Send an ACK to resynchronize and drop any data.  
     * But keep on processing for RST or ACK.  
    */  
    tp->t_flags |= TF_ACKNOW;  
    todrop = ti->ti_len;  
    tcpsstat.tcps_rcvdupbyte += todrop;  
    tcpsstat.tcps_rcvdupack++;  
  
} else {  
    tcpsstat.tcps_rcvpartduppack++;  
    tcpsstat.tcps_rcvpartdupbyte += todrop;  
}
```

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.9 Self-Connects and Simultaneous Opens

It is instructive to look at the steps involved in a socket connecting to itself to see how the one-line fix to [Figure 28.25](#) that was added to 4.4BSD allows this. This same fix allowed simultaneous opens to work, which wasn't handled correctly prior to 4.4BSD.

A process creates a socket and connects it to itself using the system calls: `socket`, `bind` a local port (say 3000), and then `connect` to this same port and some local IP address. If the `connect` succeeds, the socket is connected to itself: anything written to the socket can be read back

from the socket. This is similar to a full-duplex pipe, but with a single descriptor instead of two descriptors. Although this is of limited use within a process, we'll see that the state transitions are the same as they are for a simultaneous open. If your system doesn't allow a socket to connect to itself, it probably doesn't handle simultaneous opens correctly either, and the latter are required by RFC 1122. Some people are surprised that a self-connect even works, given that a single Internet PCB and a single TCP control block are used. But TCP is a full-duplex, symmetric protocol and it maintains separate variables for each direction of data flow.

[Figure 28.31](#) shows the send sequence space when the process calls connect. A SYN segment is sent and the state becomes SYN\_SENT.

**Figure 28.31. Send sequence space when SYN is sent for self-connect.**



The SYN is received and processed in [Figures 28.18](#) and [28.20](#), but since the SYN does not contain an ACK the resulting state is SYN\_RCVD. According to the state transition diagram ([Figure 24.15](#)), this looks like a simultaneous open. [Figure 28.32](#) shows the receive sequence space.

### **Figure 28.32. Receive sequence space after received SYN is processed.**

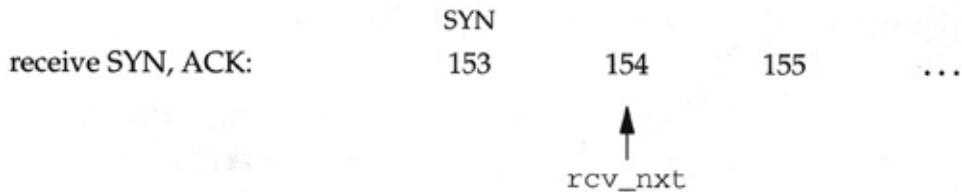


[Figure 28.20](#) sets the TF\_ACKNOW flag and the segment generated by `tcp_output` will contain a SYN and an ACK (the `tcp_outflags` value in [Figure 24.16](#)). The sequence number of the SYN is 153 and the acknowledgment number is 154.

Nothing changes in the send sequence space from [Figure 28.20](#), except the state

is now SYN\_SENT. Figure 28.33 shows the receive sequence space when the segment with the SYN and ACK is received.

### Figure 28.33. Receive sequence space when segment with SYN and ACK received.



Since the connection state is SYN\_RCVD, the segment is not processed by the active open or passive open code that we saw earlier in this chapter. It must be processed by the SYN\_RCVD code that we'll examine in Figure 29.2. But it is first processed by Figure 28.24, and it looks like a duplicate SYN:

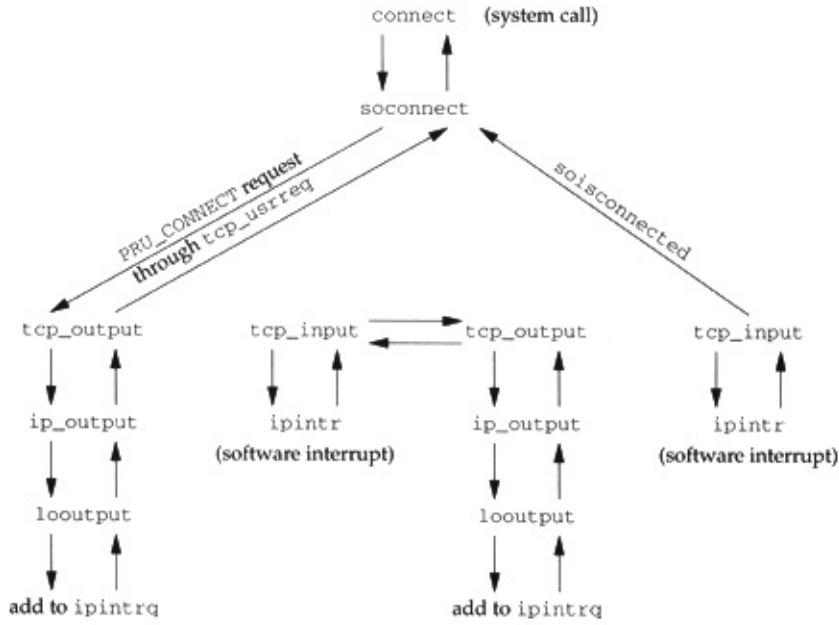
$$\begin{aligned} \text{todrop} &= \text{rcv\_nxt} - \text{ti\_seq} \\ &= 154 - 153 \\ &= 1 \end{aligned}$$

Since the SYN flag is set, the flag is

cleared, `ti_seq` becomes 154, and `todrop` becomes 0. But the test at the beginning of [Figure 28.25](#) is true, because `todrop` equals the length of the segment (0). The segment is counted as a duplicate packet and the code with the comment "Handle the case when a bound socket connects to itself" is executed. Earlier releases jumped to `dropafterack`, which skipped the necessary code to handle the `SYN_RCVD` state, preventing the connection from ever being established. Instead, Net/3 continues processing the received segment if `todrop` equals 0 and the ACK flag is set, both of which are true in this example. This allows the `SYN_RCVD` processing to happen later in the function, which moves the connection to the `ESTABLISHED` state.

It is also interesting to look at the sequence of function calls in this self-connect. This is shown in [Figure 28.34](#).

## **Figure 28.34. Sequence of function calls for self-connect.**



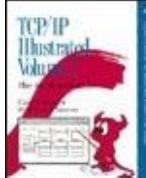
action: send SYN	process SYN	send SYN, ACK	process SYN, ACK
starting state: CLOSED	SYN_SENT	SYN_RECV	SYN_RECV
ending state: SYN_SENT	SYN_RECV	SYN_RECV	ESTABLISHED

The order of the operations goes from the left to the right. The steps that we show begin with the process calling connect. This issues the PRU\_CONNECT request, which sends a SYN down the protocol stack. Since the segment is destined for the host's own IP address it is routed to the loopback interface, which adds the segment to ipintrq and generates a software interrupt.

The software interrupt causes ipintr to execute, which calls tcp\_input. This function calls tcp\_output, causing a SYN

segment with an ACK to be sent down the protocol stack. It is again added to ipintrq by the loopback interface, and a software interrupt is generated. When this interrupt is processed by ipintr, the function `tcp_input` is called, and it moves the connection to the ESTABLISHED state.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

## 28.10 Record Timestamp

The next part of `tcp_input`, shown in [Figure 28.35](#), handles a received timestamp option.

**Figure 28.35. `tcp_input` function: record timestamp.**

```
737  /*
738   * If last ACK falls within this segment's sequence numbers,
739   * record its timestamp.
740   */
741  if (ts_present && SEQ_LBQ(ti->ti_seq, tp->last_ack_sent) &&
742      SEQ_LT(tp->last_ack_sent, ti->ti_seq + ti->ti_len +
743          (tiflags & (TH_SYN | TH_FIN)) != 0))) {
744      tp->ts_recent_age = tcp_now;
745      tp->ts_recent = ts_val;
746 }
```

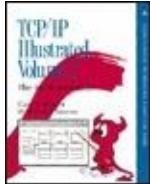
737-746

If the received segment contains a timestamp, the timestamp value is saved in `ts_recent`. We

discussed in [Section 26.6](#) how this code used by Net/3 is flawed. The expression

```
( (tiflags & (TH_SYN|TH_FIN)) !=
```

is 0 if neither of the two flags is set, or 1 if either is set. This effectively adds 1 to `ti_len` if either flag is set.



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.11 RST Processing

Figure 28.36 shows the switch statement to handle the RST flag, which depends on the connection state.

**Figure 28.36. `tcp_input` function: process RST flag.**

---

```

747     /*
748      * If the RST bit is set examine the state:
749      *   SYN RECEIVED state:
750      *   If passive open, return to LISTEN state.
751      *   If active open, inform user that connection was refused.
752      *   ESTABLISHED, FIN_WAIT_1, FIN_WAIT2, CLOSE_WAIT states:
753      *   Inform user that connection was reset, and close tcb.
754      *   CLOSING, LAST_ACK, TIME_WAIT states
755      *   Close the tcb.
756     */
757     if (tiflags & TH_RST)
758         switch (tp->t_state) {
759
760             case TCPS_SYN_RECEIVED:
761                 so->so_error = ECONNREFUSED;
762                 goto close;
763
764             case TCPS_ESTABLISHED:
765             case TCPS_FIN_WAIT_1:
766             case TCPS_FIN_WAIT_2:
767             case TCPS_CLOSE_WAIT:
768                 so->so_error = ECONNRESET;
769                 close:
770                 tp->t_state = TCPS_CLOSED;
771                 tcpstat.tcps_drops++;
772                 tp = tcp_close(tp);
773                 goto drop;
774
775             case TCPS_CLOSING:
776             case TCPS_LAST_ACK:
777             case TCPS_TIME_WAIT:
778                 tp = tcp_close(tp);
779                 goto drop;
780
781         }

```

---

tcp\_input.c

## SYN\_RCVD state

759-761

The socket's error code is set to ECONNREFUSED, and a jump is made a few lines forward to close the socket.

This state can be entered from two directions. Normally it is entered from the LISTEN state, after a SYN has been received. TCP replied with a SYN and an

ACK but received an RST in reply. Perhaps the other end sent its SYN and then terminated before the reply arrived, causing it to send an RST. In this case the socket referred to by so is the new socket created by sonewconn in [Figure 28.7](#). Since dropsocket will still be true, the socket is discarded at the label drop. The listening descriptor isn't affected at all. This is why we show the state transition from SYN\_RCVD back to LISTEN in [Figure 24.15](#).

This state can also be entered by a simultaneous open, after a process has called connect. In this case the socket error is returned to the process.

## Other states

762-777

The receipt of an RST in the ESTABLISHED, FIN\_WAIT\_1, FIN\_WAIT\_2, or CLOSE\_WAIT states returns the error ECONNRESET. In the CLOSING, LAST\_ACK, and TIME\_WAIT state an error is not

generated, since the process has closed the socket.

Allowing an RST to terminate a connection in the TIME\_WAIT state circumvents the reason this state exists. RFC 1337 [Braden 1992] discusses this and other forms of "TIME\_WAIT assassination hazards" and recommends *not* letting an RST prematurely terminate the TIME\_WAIT state. See [Exercise 28.10](#) for an example.

The next piece of code, shown in [Figure 28.37](#), checks for erroneous SYNs and verifies that an ACK is present.

### Figure 28.37. `tcp_input` function: handle SYN-full and ACK-less segments.

```
778  /*
779   * If a SYN is in the window, then this is an
780   * error and we send an RST and drop the connection.
781   */
782  if (tiflags & TH_SYN) {
783      tp = tcp_drop(tp, ECONNRESET);
784      goto dropwithreset;
785  }
786  /*
787   * If the ACK bit is off we drop the segment and return.
788   */
789  if ((tiflags & TH_ACK) == 0)
790      goto drop;
```

778-785

If the SYN flag is still set, this is an error and the connection is dropped with the error ECONNRESET.

786-790

If the ACK flag is not set, the segment is dropped. The remainder of this function, which we continue in the next chapter, assumes the ACK flag is set.

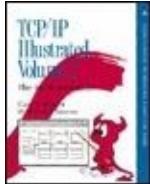
---

Team-Fly

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 28. TCP Input

### 28.12 Summary

This chapter has started our detailed look at TCP input. It continues in the next chapter.

The code in this chapter verifies the segment's checksum, processes any TCP options, handles SYNs that initiate or complete connection requests, trims excess data from the beginning or end of the segment, and processes the RST flag.

Header prediction is a successful attempt to handle common cases with the minimum amount of processing. Although the general processing steps that we've covered handle all possible cases (which

they must), many segments are well behaved and the processing steps can be minimized.

## Exercises

Given that the maximum size of a socket buffer is 262,144 in Net/3,

**28.1** what are the possible window scale shift factors calculated by [Figure 28.7](#)?

Given that the maximum size of a socket buffer is 262,144 in Net/3, what is the maximum throughput

**28.2** possible with a round-trip time of 60 ms? (*Hint:* See Figure 24.5 in Volume 1 and solve for the bandwidth.)

Why are the two timestamp values

**28.3** fetched using bcopy in [Figure 28.10](#)?

We mentioned in [Section 26.6](#) that TCP correctly handles timestamp options in a format other than the **28.4** one recommended in Appendix A of RFC 1323. While this is true, what is the penalty for not following the recommended format?

The PRU\_ATTACH request allocates the PCB and the TCP control block, but doesn't call `tcp_template` to allocate the header template.

**28.5** Instead we saw in [Figure 28.17](#) that the header template is allocated when the SYN arrives. Why doesn't the PRU\_ATTACH request allocate this template?

Read RFC 1323 to determine why **28.6** the limit of 24 days was chosen in [Figure 28.22](#).

The comparison of `tcp_now` minus `ts_recent_age` to `TCP_PAWS_IDLE` in [Figure 28.22](#) is also subject to

**28.7** sign bit wrap around, if the connection is idle for a period much longer than 24 days. With the 500-ms timestamp clock used by Net/3, when does this become a problem?

Read RFC 1323 to find out why RST  
**28.8** segments are exempt from the PAWS test in [Figure 28.22](#).

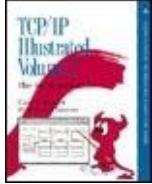
**28.9** A client sends a SYN and the server responds with a SYN/ACK. The client moves to the ESTABLISHED state and responds with an ACK, but this ACK is lost. The server resends its SYN/ACK. Describe the processing steps when the client receives this duplicate SYN/ACK.

A client and server have an established connection and the server performs the active close. The connection terminates normally and the socket pair goes into the TIME\_WAIT state on the server.

Before this 2MSL wait expires on **28.10** the server, the same client (i.e., the same socket pair on the client) sends a SYN to the server's socket pair but with a sequence number that is less than the ending sequence number from the previous incarnation of this connection. Describe what happens.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 29. TCP Input (Continued)

Section 29.1. Introduction

Section 29.2. ACK Processing Overview

Section 29.3. Completion of Passive  
Opens and Simultaneous Opens

Section 29.4. Fast Retransmit and Fast  
Recovery Algorithms

Section 29.5. ACK Processing

Section 29.6. Update Window  
Information

Section 29.7. Urgent Mode Processing

Section 29.8. `tcp_pullofband`  
Function

Section 29.9. Processing of Received  
Data

[Section 29.10. FIN Processing](#)

[Section 29.11. Final Processing](#)

[Section 29.12. Implementation  
Refinements](#)

[Section 29.13. Header Compression](#)

[Section 29.14. Summary](#)

---

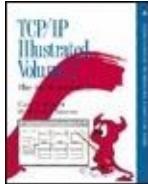
**Team-Fly**



[Previous](#)

[Next](#)

[Top](#)



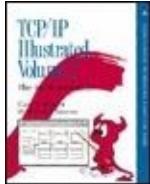
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.1 Introduction

This chapter continues the discussion of TCP input processing, picking up where the previous chapter left off. Recall that the final test in [Figure 28.37](#) was that either the ACK flag was set or, if not, the segment was dropped.

The ACK flag is handled, the window information is updated, the URG flag is processed, and any data in the segment is processed. Finally the FIN flag is processed and `tcp_output` is called, if required.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

## 29.2 ACK Processing Overview

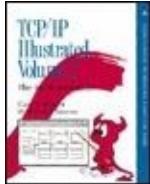
We begin this chapter with ACK processing, a summary of which is shown in [Figure 29.1](#). The SYN\_RCVD state is handled specially, followed by common processing for all remaining states. (Remember that a received ACK in either the LISTEN or SYN\_SENT state was discussed in the previous chapter.) This is followed by special processing for the three states in which a received ACK causes a state transition, and for the TIME\_WAIT state, in which the receipt of an ACK causes the 2MSL timer to be restarted.

## Figure 29.1. Summary of ACK processing.

---

```
switch (tp->t_state) {  
  
    case TCPS_SYN RECEIVED:  
        complete processing of passive open and process  
        simultaneous open or self-connect;  
        /* fall into ... */  
  
    case TCPS_ESTABLISHED:  
    case TCPS_FIN_WAIT_1:  
    case TCPS_FIN_WAIT_2:  
    case TCPS_CLOSE_WAIT:  
    case TCPS_CLOSING:  
    case TCPS_LAST_ACK:  
    case TCPS_TIME_WAIT:  
        process duplicate ACK;  
        update RTT estimators;  
        if all outstanding data ACKed, turn off retransmission timer;  
        remove ACKed data from socket send buffer;  
        switch (tp->t_state) {  
            case TCPS_FIN_WAIT_1:  
                if (FIN is ACKed) {  
                    move to FIN_WAIT_2 state;  
                    start FIN_WAIT_2 timer;  
                }  
                break;  
            case TCPS_CLOSING:  
                if (FIN is ACKed) {  
                    move to TIME_WAIT state;  
                    start TIME_WAIT timer;  
                }  
                break;  
            case TCPS_LAST_ACK:  
                if (FIN is ACKed)  
                    move to CLOSED state;  
                break;  
            case TCPS_TIME_WAIT:  
                restart TIME_WAIT timer;  
                goto dropafterack;  
        }  
    }  
}
```

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.3 Completion of Passive Opens and Simultaneous Opens

The first part of the ACK processing, shown in [Figure 29.2](#), handles the SYN\_RCVD state. As mentioned in the previous chapter, this handles the completion of a passive open (the common case) and also handles simultaneous opens and self-connects (the infrequent case).

**Figure 29.2. `tcp_input` function: received ACK in SYN\_RCVD state.**

```

791  /*
792   * Ack processing.
793   */
794  switch (tp->t_state) {
795
796      /*
797       * In SYN RECEIVED state if the ack ACKs our SYN then enter
798       * ESTABLISHED state and continue processing, otherwise
799       * send an RST.
800      */
800  case TCPS_SYN_RECEIVED:
801      if (SEQ_GT(tp->snd_una, ti->ti_ack) ||
802          SEQ_GT(ti->ti_ack, tp->snd_max))
803          goto dropwithreset;
804      tcpstat.tcp_connects++;
805      soisconnected(so);
806      tp->t_state = TCPS_ESTABLISHED;
807      /* Do window scaling? */
808      if ((tp->t_flags & (TF_RCVD_SCALE | TF_REQ_SCALE)) ==
809          (TF_RCVD_SCALE | TF_REQ_SCALE)) {
810          tp->snd_scale = tp->requested_s_scale;
811          tp->rcv_scale = tp->request_r_scale;
812      }
813      (void) tcp_reass(tp, (struct tciphdr *) 0, (struct mbuf *) 0);
814      tp->snd_w1 = ti->ti_seq - 1;
815      /* fall into ... */

```

tcp\_input.c

## Verify received ACK

*801-806*

For the ACK to acknowledge the SYN that was sent, it must be greater than snd\_una (which is set to the ISS for the connection, the sequence number of the SYN, by `tcp_sendseqinit`) and less than or equal to snd\_max. If so, the socket is marked as connected and the state becomes ESTABLISHED.

Since `soisconnected` wakes up the process that performed the passive open (normally a server), we see that this doesn't occur

until the last of the three segments in the three-way handshake has been received. If the server is blocked in a call to accept, that call now returns; if the server is blocked in a call to select waiting for the listening descriptor to become readable, it is now readable.

## Check for window scale option

807-812

If TCP sent a window scale option and received one, the send and receive scale factors are saved in the TCP control block. Otherwise the default values of `snd_scale` and `rcv_scale` in the TCP control block are 0 (no scaling).

## Pass queued data to process

813

Any data queued for the connection can now be passed to the process. This is done by `tcp_reass` with a null pointer as the second argument. This data would have

arrived with the SYN that moved the connection into the SYN\_RCVD state.

814

snd\_wl1 is set to the received sequence number minus 1. We'll see in [Figure 29.15](#) that this causes the three window update variables to be updated.

---

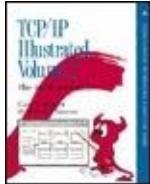
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.4 Fast Retransmit and Fast Recovery Algorithms

The next part of ACK processing, shown in [Figure 29.3](#), handles duplicate ACKs and determines if TCP's fast retransmit and fast recovery algorithms [[Jacobson 1990c](#)] should come into play. The two algorithms are separate but are normally implemented together [[Floyd 1994](#)].

**Figure 29.3. `tcp_input` function: check for completely duplicate ACK.**

```

816     /*
817      * In ESTABLISHED state: drop duplicate ACKs; ACK out-of-range
818      * ACKs. If the ack is in the range
819      * tp->snd_una < ti->ti_ack <= tp->snd_max
820      * then advance tp->snd_una to ti->ti_ack and drop
821      * data from the retransmission queue. If this ACK reflects
822      * more up-to-date window information we update our window information.
823      */
824     case TCPS_ESTABLISHED:
825     case TCPS_FIN_WAIT_1:
826     case TCPS_FIN_WAIT_2:
827     case TCPS_CLOSE_WAIT:
828     case TCPS_CLOSING:
829     case TCPS_LAST_ACK:
830     case TCPS_TIME_WAIT:
831
832         if (SEQ_LEQ(ti->ti_ack, tp->snd_una)) {
833             if (ti->ti_len == 0 && tiwin == tp->snd_wnd) {
834                 tcpstat.tcpstat_rcvdupack++;
835                 /*
836                  * If we have outstanding data (other than
837                  * a window probe), this is a completely
838                  * duplicate ack (ie, window info didn't
839                  * change), the ack is the biggest we've
840                  * seen and we've seen exactly our rexmt
841                  * threshold of them, assume a packet
842                  * has been dropped and retransmit it.
843                  * Kludge snd_nxt & the congestion
844                  * window so we send only this one
845                  * packet.
846                  *
847                  * We know we're losing at the current
848                  * window size so do congestion avoidance
849                  * (set ssthresh to half the current window
850                  * and pull our congestion window back to
851                  * the new ssthresh).
852                  *
853                  * Dup acks mean that packets have left the
854                  * network (they're now cached at the receiver)
855                  * so bump cwnd by the amount in the receiver
856                  * to keep a constant cwnd packets in the
857                  * network.
858             */

```

tcp\_input.c

- The *fast retransmit* algorithm occurs when TCP deduces from a small number (normally 3) of consecutive duplicate ACKs that a segment has been lost and deduces the starting sequence number of the missing segment. The missing segment is retransmitted. The algorithm is mentioned in Section 4.2.2.21 of RFC 1122, which states that TCP may generate an immediate ACK

when an out-of-order segment is received. We saw that Net/3 generates the immediate duplicate ACKs in [Figure 27.15](#). This algorithm first appeared in the 4.3BSD Tahoe release and the subsequent Net/1 release. In these two implementations, after the missing segment was retransmitted, the slow start phase was entered.

- The *fast recovery* algorithm says that after the fast retransmit algorithm (that is, after the missing segment has been retransmitted), congestion avoidance but not slow start is performed. This is an improvement that allows higher throughput under moderate congestion, especially for large windows. This algorithm appeared in the 4.3BSD Reno release and the subsequent Net/2 release.

Net/3 implements both fast retransmit and fast recovery, as we describe shortly.

In the discussion of [Figure 24.17](#) we noted that an acceptable ACK must be in the range

`snd_una < acknowledgment field <= snd_max`

This first test of the acknowledgment field compares it only to `snd_una`. The comparison against `snd_max` is in [Figure 29.5](#). The reason for separating the tests is so that the following five tests can be applied to the received segment:

**1. If the acknowledgment field is less than or equal to `snd_una`, and**

- the length of the received segment is 0, and
- the advertised window (`tiwin`) has not changed, and
- TCP has outstanding data that has not been acknowledged (the retransmission timer is nonzero), and
- the received segment contains the biggest ACK TCP has seen (the acknowledgment field equals `snd_una`),

then this segment is a completely duplicate ACK. (Tests 1, 2, and 3 are in

Figure 29.3; tests 4 and 5 are at the beginning of Figure 29.4.)

## Figure 29.4. `tcp_input` function: duplicate ACK processing.

```
858             if (tp->t_timer[TCPT_REXMT] == 0 ||  
859                 ti->ti_ack != tp->snd_una)  
860                 tp->t_dupacks = 0;  
861             else if (++tp->t_dupacks == tcprexmtthresh) {  
862                 tcp_seq onxt = tp->snd_nxt;  
863                 u_int win =  
864                     min(tp->snd_wnd, tp->snd_cwnd) / 2 /  
865                     tp->t_maxseg;  
866  
867                 if (win < 2)  
868                     win = 2;  
869                 tp->snd_ssthresh = win * tp->t_maxseg;  
870                 tp->t_timer[TCPT_REXMT] = 0;  
871                 tp->t_rtt = 0;  
872                 tp->snd_nxt = ti->ti_ack;  
873                 tp->snd_cwnd = tp->t_maxseg;  
874                 (void) tcp_output(tp);  
875                 tp->snd_cwnd = tp->snd_ssthresh +  
876                     tp->t_maxseg * tp->t_dupacks;  
877                 if (SEQ_GT(onxt, tp->snd_nxt))  
878                     tp->snd_nxt = onxt;  
879                 goto drop;  
880             } else if (tp->t_dupacks > tcprexmtthresh) {  
881                 tp->snd_cwnd += tp->t_maxseg;  
882                 (void) tcp_output(tp);  
883                 goto drop;  
884             }  
885         } else  
886             tp->t_dupacks = 0;  
887         break; /* beyond ACK processing (to step 6) */  
888     }
```

*tcp\_input.c*

TCP counts the number of these duplicate ACKs that are received in a row (in the variable `t_dupacks`), and when the number reaches a threshold of 3 (`tcprexmtthresh`), the lost segment is retransmitted. This is the *fast retransmit* algorithm described in Section 21.7 of Volume 1. It works in

conjunction with the code we saw in [Figure 27.15](#): when TCP receives an out-of-order segment, it is required to generate an immediate duplicate ACK, telling the other end that a segment might have been lost and telling it the value of the next expected sequence number. The goal of the fast retransmit algorithm is for TCP to retransmit immediately what appears to be the missing segment, instead of waiting for the retransmission timer to expire. Figure 21.7 of Volume 1 gives a detailed example of how this algorithm works.

The receipt of a duplicate ACK also tells TCP that a packet has "left the network," because the other end had to receive an out-of-order segment to send the duplicate ACK. The *fast recovery* algorithm says that after some number of consecutive duplicate ACKs have been received, TCP should perform congestion avoidance (i.e., slow down) but need not wait for the pipe to empty between the two connection end points (slow start). The expression "a packet has left the network" means a packet has been received by the other end and has been added to the out-of-order

queue for the connection. The packet is not still in transit somewhere between the two end points.

If only the first three tests shown earlier are true, the ACK is still a duplicate and is counted by the statistic `tcps_rcvdu` pack, but the counter of the number of consecutive duplicate ACKs for this connection (`t_dupacks`) is reset to 0. If only the first test is true, the counter `t_dupacks` is reset to 0.

The remainder of the fast recovery algorithm is shown in [Figure 29.4](#). When all five tests are true, the fast recovery algorithm processes the segment depending on the number of these consecutive duplicate ACKs that have been received.

- 1. `t_dupacks` equals 3  
(`tcpreno` `thresh`). Congestion avoidance is performed and the missing segment is retransmitted.**
  - `t_dupacks` exceeds 3. Increase the congestion window and perform normal

TCP output.

- t\_dupacks is less than 3. Do nothing.

## Number of consecutive duplicate ACKs reaches threshold of 3

861-868

When t\_dupacks reaches 3 (tcp\_rexmtthresh), the value of snd\_nxt is saved in onxt and the slow start threshold (ssthresh) is set to one-half the current congestion window, with a minimum value of two segments. This is what was done with the slow start threshold when the retransmission timer expired in [Figure 25.27](#), but we'll see later in this piece of code that the fast recovery algorithm does not set the congestion window to one segment, as was done with the timeout.

## Turn off retransmission timer

869-870

The retransmission timer is turned off and,

in case a segment is currently being timed, `t_rtt` is set to 0.

## Retransmit missing segment

871-873

`snd_nxt` is set to the starting sequence number of the segment that appears to have been lost (the acknowledgment field of the duplicate ACK) and the congestion window is set to one segment. This causes `tcp_output` to send only the missing segment. (This is shown by segment 63 in Figure 21.7 of Volume 1.)

## Set congestion window

874-875

The congestion window is set to the slow start threshold plus the number of segments that the other end has cached. By *cached* we mean the number of out-of-order segments that the other end has received and generated duplicate ACKs for. These cannot be passed to the process at

the other end until the missing segment (which was just sent) is received. Figures 21.10 and 21.11 in Volume 1 show what happens with the congestion window and slow start threshold when the fast recovery algorithm is in effect.

## **Set snd\_nxt**

876-878

The value of the next sequence number to send is set to the maximum of its previous value (onxt) and its current value. Its current value was modified by tcp\_output when the segment was retransmitted. Normally this causes snd\_nxt to be set back to its previous value, which means that only the missing segment is retransmitted, and that future calls to tcp\_output carry on with the next segment in sequence.

## **Number of consecutive duplicate ACKs exceeds threshold of 3**

879-883

The missing segment was retransmitted when t\_dupacks equaled 3, so the receipt of each additional duplicate ACK means that another packet has left the network. The congestion window is incremented by one segment. tcp\_output sends the next segment in sequence, and the duplicate ACK is dropped. (This is shown by segments 67, 69, and 71 in Figure 21.7 of Volume 1.)

884-885

This statement is executed when the received segment contains a duplicate ACK, but either the length is nonzero or the advertised window changed. Only the first of the five tests described earlier is true. The counter of consecutive duplicate ACKs is set to 0.

## Skip remainder of ACK processing

886

This break is executed in three cases: (1)

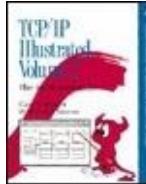
only the first of the five tests described earlier is true, or (2) only the first three of the five tests is true, or (3) the ACK is a duplicate, but the number of consecutive duplicates is less than the threshold of 3. For any of these cases the ACK is still a duplicate and the break goes to the end of the switch that started in [Figure 29.2](#), which continues processing at the label step6.

To understand the purpose in this aggressive window manipulation, consider the following example. Assume the window is eight segments, and segments 1 through 8 are sent. Segment 1 is lost, but the remainder arrive OK and are acknowledged. After the ACKs for segments 2, 3, and 4 arrive, the missing segment (1) is retransmitted. TCP would like the subsequent ACKs for 5 through 8 to allow some of the segments starting with 9 to be sent, to keep the pipe full. But the window is 8, which prevents segments 9 and above from being sent. Therefore, the congestion window is temporarily inflated by one segment each time another duplicate ACK is received, since the receipt

of the duplicate ACK tells TCP that another segment has left the pipe at the other end. When the acknowledgment of segment 1 is finally received, the next figure reduces the congestion window back to the slow start threshold. This increase in the congestion window as the duplicate ACKs arrive, and its subsequent decrease when the fresh ACK arrives, can be seen visually in Figure 21.10 of Volume 1.

---





TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

## 29.5 ACK Processing

The ACK processing continues with [Figure 29.5](#).

### Figure 29.5. `tcp_input` function: ACK processing continued.

```
888     /*                                         -tcp_input.c
889     * If the congestion window was inflated to account
890     * for the other side's cached packets, retract it.
891     */
892     if (tp->t_dupacks > tcprexmtthresh &&
893         tp->snd_cwnd > tp->snd_ssthresh)
894         tp->snd_cwnd = tp->snd_ssthresh;
895     tp->t_dupacks = 0;
896
897     if (SEQ_GT(ti->ti_ack, tp->snd_max)) {
898         tcpstat.tcpst_rcvacktoomuch++;
899         goto dropafterack;
900     }
901     acked = ti->ti_ack - tp->snd_una;
902     tcpstat.tcpst_rcvackpack++;
903     tcpstat.tcpst_rcvackbyte += acked;
```

## Adjust congestion window

888-895

If the number of consecutive duplicate ACKs exceeds the threshold of 3, this is the first nonduplicate ACK after a string of four or more duplicate ACKs. The fast recovery algorithm is complete. Since the congestion window was incremented by one segment for every consecutive duplicate after the third, if it now exceeds the slow start threshold, it is set back to the slow start threshold. The counter of consecutive duplicate ACKs is set to 0.

## Check for out-of-range ACK

896-899

Recall the definition of an acceptable ACK,

$\text{snd\_una} < \text{acknowledgment field} \leq \text{snd\_max}$

If the acknowledgment field is greater than  $\text{snd\_max}$ , the other end is acknowledging

data that TCP hasn't even sent yet! This probably occurs on a high-speed connection when the sequence numbers wrap and a missing ACK reappears later. As we can see in [Figure 24.5](#), this rarely happens (since today's networks aren't fast enough).

## Calculate number of bytes acknowledged

900-902

At this point TCP knows that it has an acceptable ACK. `acked` is the number of bytes acknowledged.

The next part of ACK processing, shown in [Figure 29.6](#), deals with RTT measurements and the retransmission timer.

**Figure 29.6. `tcp_input` function: RTT measurements and retransmission timer.**

```

903     /*
904      * If we have a timestamp reply, update smoothed
905      * round-trip time. If no timestamp is present but
906      * transmit timer is running and timed sequence
907      * number was acked, update smoothed round-trip time.
908      * Since we now have an rtt measurement, cancel the
909      * timer backoff (cf., Phil Karn's retransmit alg.).
910      * Recompute the initial retransmit timer.
911     */
912     if (ts_present)
913         tcp_xmit_timer(tp, tcp_now - ts_ecr + 1);
914     else if (tp->t_rtt && SEQ_GT(ti->ti_ack, tp->t_rtseq))
915         tcp_xmit_timer(tp, tp->t_rtt);

916     /*
917      * If all outstanding data is acked, stop retransmit
918      * timer and remember to restart (more output or persist).
919      * If there is more data to be acked, restart retransmit
920      * timer, using current (possibly backed-off) value.
921     */
922     if (ti->ti_ack == tp->snd_max) {
923         tp->t_timer[TCPT_REXMT] = 0;
924         needoutput = 1;
925     } else if (tp->t_timer[TCPT_PERSIST] == 0)
926         tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;

```

tcp\_input.c

## Update RTT estimators

903-915

If either (1) a timestamp option was present, or (2) a segment was being timed and the acknowledgment number is greater than the starting sequence number of the segment being timed, `tcp_xmit_timer` updates the RTT estimators. Notice that the second argument to this function when timestamps are used is the current time (`tcp_now`) minus the timestamp echo reply (`ts_ecr`) plus 1 (since the function subtracts 1).

Delayed ACKs are the reason for the greater-than test of the sequence numbers. For example, if TCP sends and times a segment with bytes 11024, followed by a segment with bytes 10252048, if an ACK of 2049 is returned, this test will consider whether 2049 is greater than 1 (the starting sequence number of the segment being timed), and since this is true, the RTT estimators are updated.

## Check if all outstanding data has been acknowledged

916-924

If the acknowledgment field of the received segment (`ti_ack`) equals the maximum sequence number that TCP has sent (`snd_max`), all outstanding data has been acknowledged. The retransmission timer is turned off and the `needoutput` flag is set to 1. This flag forces a call to `tcp_output` at the end of this function. Since there is no more data waiting to be acknowledged, TCP may have more data to send that it

has not been able to send earlier because the data was beyond the right edge of the window. Now that a new ACK has been received, the window will probably move to the right (`snd_una` is updated in [Figure 29.8](#)), which could allow more data to be sent.

## Unacknowledged data outstanding

925-926

Since there is additional data that has been sent but not acknowledged, if the persist timer is not on, the retransmission timer is restarted using the current value of `t_rxtcur`.

## Karn's Algorithm and Timestamps

Notice that timestamps overrule the portion of Karn's algorithm (Section 21.3 of Volume 1) that says: when a timeout and retransmission occurs, the RTT estimators cannot be updated when the acknowledgment for the retransmitted data is received (the *retransmission ambiguity*)

*problem*). In [Figure 25.26](#) we saw that `t_rtt` was set to 0 when a retransmission took place, because of Karn's algorithm. If timestamps are not present and it is a retransmission, the code in [Figure 29.6](#) does not update the RTT estimators because `t_rtt` will be 0 from the retransmission. But if a timestamp is present, `t_rtt` isn't examined, allowing the RTT estimators to be updated using the received timestamp echo reply. With RFC 1323 timestamps the ambiguity is gone since the `ts_ecr` value was copied by the other end from the segment being acknowledged. The other half of Karn's algorithm, specifying that an exponential backoff must be used with retransmissions, still holds, of course.

[Figure 29.7](#) shows the next part of ACK processing, updating the congestion window.

**Figure 29.7. `tcp_input` function: open congestion window in response to ACKs.**

---

```

927     /*
928      * When new data is acked, open the congestion window.
929      * If the window gives us less than ssthresh packets
930      * in flight, open exponentially (maxseg per packet).
931      * Otherwise open linearly: maxseg per window
932      * (maxseg^2 / cwnd per packet), plus a constant
933      * fraction of a packet (maxseg/8) to help larger windows
934      * open quickly enough.
935      */
936 {
937     u_int    cw = tp->snd_cwnd;
938     u_int    incr = tp->t_maxseg;
939
940     if (cw > tp->snd_ssthresh)
941         incr = incr * incr / cw + incr / 8;
942     tp->snd_cwnd = min(cw + incr, TCP_MAXWIN << tp->snd_scale);
943 }

```

---

*tcp\_input.c*

## Update congestion window

927-942

One of the rules of slow start and congestion avoidance is that a received ACK increases the congestion window. By default the congestion window is increased by one segment for each received ACK (slow start). But if the current congestion window is greater than the slow start threshold, it is increased by 1 divided by the congestion window, plus a constant fraction of a segment. The term

incr \* incr / cw

is

`t_maxseg * t_maxseg / snd_cwnd`

which is 1 divided by the congestion window, taking into account that `snd_cwnd` is maintained in bytes, not segments. The constant fraction is the segment size divided by 8. The congestion window is then limited by the maximum value of the send window for this connection. Example calculations of this algorithm are in Section 21.8 of Volume 1.

Adding in the constant fraction (the segment size divided by 8) is wrong [[Floyd 1994](#)]. But it has been in the BSD sources since 4.3BSD Reno and is still in 4.4BSD and Net/3. It should be removed.

The next part of `tcp_input`, shown in [Figure 29.8](#), removes the acknowledged data from the send buffer.

**Figure 29.8. `tcp_input` function: remove acknowledged data from send buffer.**

```

943     if (acked > so->so_snd.sb_cc) {
944         tp->snd_wnd -= so->so_snd.sb_cc;
945         sbdrop(&so->so_snd, (int) so->so_snd.sb_cc);
946         ourfinisacked = 1;
947     } else {
948         sbdrop(&so->so_snd, acked);
949         tp->snd_wnd -= acked;
950         ourfinisacked = 0;
951     }
952     if (so->so_snd.sb_flags & SB_NOTIFY)
953         sowakeup(so);
954     tp->snd_una = ti->ti_ack;
955     if (SEQ_LT(tp->snd_nxt, tp->snd_una))
956         tp->snd_nxt = tp->snd_una;

```

*tcp\_input.c*

## Remove acknowledged bytes from the send buffer

**943-946**

If the number of bytes acknowledged exceeds the number of bytes on the send buffer, `snd_wnd` is decremented by the number of bytes in the send buffer and TCP knows that its FIN has been ACKed. That number of bytes is then removed from the send buffer by `sbdrop`. This method for detecting the ACK of a FIN works only because the FIN occupies 1 byte in the sequence number space.

**947-951**

Otherwise the number of bytes acknowledged is less than or equal to the

number of bytes in the send buffer, so ourfinisacked is set to 0, and acked bytes of data are dropped from the send buffer.

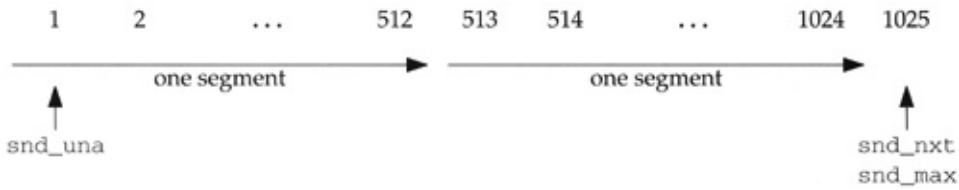
## **Wakeup processes waiting on send buffer**

951-956

sowakeup awakens any processes waiting on the send buffer. snd\_una is updated to contain the oldest unacknowledged sequence number. If this new value of snd\_una exceeds snd\_nxt, the latter is updated, since the intervening bytes have been acknowledged.

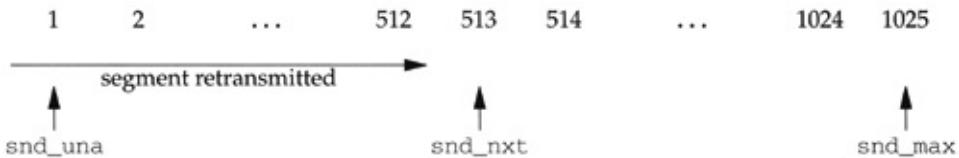
[Figure 29.9](#) shows how snd\_nxt can end up with a sequence number that is less than snd\_una. Assume two segments are transmitted, the first with bytes 1512 and the second with bytes 5131024.

**Figure 29.9. Two segments sent on a connection.**



The retransmission timer then expires before an acknowledgment is returned. The code in [Figure 25.26](#) sets `snd_nxt` back to `snd_una`, slow start is entered, `tcp_output` is called, and one segment containing bytes 1512 is retransmitted. `tcp_output` increases `snd_nxt` to 513, and we have the scenario shown in [Figure 29.10](#).

**Figure 29.10. Continuation of Figure 29.9 after retransmission timer expires.**



At this point an ACK of 1025 arrives (either the two original segments or the ACK was delayed somewhere in the network). The ACK is valid since it is less than or equal to `snd_max`, but `snd_nxt` will be less than the updated value of `snd_una`.

The general ACK processing is now complete, and the switch shown in [Figure 29.11](#) handles four special cases.

## Figure 29.11. `tcp_input` function: receipt of ACK in `FIN_WAIT_1` state.

```
957     switch (tp->t_state) {                                     tcp_input.c
958         /*
959          * In FIN_WAIT_1 state in addition to the processing
960          * for the ESTABLISHED state if our FIN is now acknowledged
961          * then enter FIN_WAIT_2.
962          */
963     case TCPS_FIN_WAIT_1:
964         if (ourfinisacked) {
965             /*
966              * If we can't receive any more
967              * data, then closing user can proceed.
968              * Starting the timer is contrary to the
969              * specification, but if we don't get a FIN
970              * we'll hang forever.
971              */
972             if (so->so_state & SS_CANTRCVMORE) {
973                 soisdisconnected(so);
974                 tp->t_timer[TCPT_2MSL] = tcp_maxidle;
975             }
976             tp->t_state = TCPS_FIN_WAIT_2;
977         }
978     break;
```

## Receipt of ACK in `FIN_WAIT_1` state

958-971

In this state the process has closed the connection and TCP has sent the FIN. But other ACKs can be received for data segments sent before the FIN. Therefore

the connection moves into the FIN\_WAIT\_2 state only when the FIN has been acknowledged. The flag ourfinisacked is set in [Figure 29.8](#); this depends on whether the number of bytes ACKed exceeds the amount of data in the send buffer or not.

## **Set FIN\_WAIT\_2 timer**

972-975

We also described in [Section 25.6](#) how Net/3 sets a FIN\_WAIT\_2 timer to prevent an infinite wait in the FIN\_WAIT\_2 state. This timer is set only if the process completely closed the connection (i.e., the close system call or its kernel equivalent if the process was terminated by a signal), and not if the process performed a half-close (i.e., the FIN was sent but the process can still receive data on the connection).

[Figure 29.12](#) shows the receipt of an ACK in the CLOSING state.

**Figure 29.12. `tcp_input` function: receipt**

## of ACK in CLOSING state.

```
tcp_input.c
979      /*
980      * In CLOSING state in addition to the processing for
981      * the ESTABLISHED state if the ACK acknowledges our FIN
982      * then enter the TIME_WAIT state, otherwise ignore
983      * the segment.
984      */
985     case TCPS_CLOSING:
986     if (ourfinisacked) {
987         tp->t_state = TCPS_TIME_WAIT;
988         tcp_canceltimers(tp);
989         tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
990         soisdisconnected(so);
991     }
992     break;
```

---

tcp\_input.c

## Receipt of ACK in CLOSING state

979-992

If the ACK is for the FIN (and not for some previous data segment), the connection moves into the TIME\_WAIT state. Any pending timers are cleared (such as a pending retransmission timer), and the TIME\_WAIT timer is started with a value of twice the MSL.

The processing of an ACK in the LAST\_ACK state is shown in [Figure 29.13](#).

**Figure 29.13. `tcp_input` function: receipt of ACK in LAST\_ACK state.**

```

993      /*
994      * In LAST_ACK, we may still be waiting for data to drain
995      * and/or to be acked, as well as for the ack of our FIN.
996      * If our FIN is now acknowledged, delete the TCB,
997      * enter the closed state, and return.
998      */
999     case TCPS_LAST_ACK:
1000        if (ourfinisacked) {
1001            tp = tcp_close(tp);
1002            goto drop;
1003        }
1004        break;

```

tcp\_input.c

## Receipt of ACK in LAST\_ACK state

993-1004

If the FIN is ACKed, the new state is CLOSED. This state transition is handled by `tcp_close`, which also releases the Internet PCB and TCP control block.

[Figure 29.14](#) shows the processing of an ACK in the TIME\_WAIT state.

## Figure 29.14. `tcp_input` function: receipt of ACK in TIME\_WAIT state.

```

1005      /*
1006      * In TIME_WAIT state the only thing that should arrive
1007      * is a retransmission of the remote FIN. Acknowledge
1008      * it and restart the finack timer.
1009      */
1010     case TCPS_TIME_WAIT:
1011        tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1012        goto dropafterack;
1013    }
1014 }

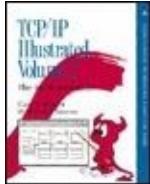
```

tcp\_input.c

## Receipt of ACK in TIME\_WAIT state

*1005-1014*

In this state both ends have sent a FIN and both FINs have been acknowledged. If TCP's ACK of the remote FIN was lost, however, the other end will retransmit the FIN (with an ACK). TCP drops the segment and resends the ACK. Additionally, the TIME\_WAIT timer must be restarted with a value of twice the MSL.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

## 29.6 Update Window Information

There are two variables in the TCP control block that we haven't described yet: `snd_wl1` and `snd_wl2`.

- `snd_wl1` records the sequence number of the last segment used to update the send window (`snd_wnd`).
- `snd_wl2` records the acknowledgment number of the last segment used to update the send window.

Our only encounter with these variables so far was when a connection was established (active, passive, or simultaneous open) and `snd_wl1` was set to `ti_seq` minus 1. We said this was to guarantee a window

update, which we'll see in the following code.

The send window (snd\_wnd) is updated from the advertised window in the received segment (tiwin) if any one of the following three conditions is true:

**1. The segment contains new data.**

**Since `snd_wl1` contains the starting sequence number of the last segment that was used to update the send window, if**

`snd_wl1 < ti_seq`

**this condition is true.**

- The segment does not contain new data (`snd_wl1` equals `ti_seq`), but the segment acknowledges new data. The latter condition is true if

`snd_wl2 < ti_ack`

since `snd_wl2` records the acknowledgment number of the last segment that updated the send window.

- The segment does not contain new data, and the segment does not acknowledge new data, but the advertised window is larger than the current send window.

The purpose of these tests is to prevent an old segment from affecting the send window, since the send window is not an absolute sequence number, but is an offset from `snd_una`.

[Figure 29.15](#) shows the code that implements the update of the send window.

**Figure 29.15. `tcp_input` function: update window information.**

```

1015     step6:
1016     /*
1017      * Update window information.
1018      * Don't look at window if no ACK: TAC's send garbage on first SYN.
1019      */
1020     if ((tiflags & TH_ACK) &&
1021         (SEQ_LT(tp->snd_wl1, ti->ti_seq) || tp->snd_wl1 == ti->ti_seq &&
1022          (SEQ_LT(tp->snd_wl2, ti->ti_ack) ||
1023           tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd))) {
1024         /* keep track of pure window updates */
1025         if (ti->ti_len == 0 &&
1026             tp->snd_wl2 == ti->ti_ack && tiwin > tp->snd_wnd)
1027             tcpstat.tcpstat_rcvwinupd++;
1028
1029         tp->snd_wnd = tiwin;
1030         tp->snd_wl1 = ti->ti_seq;
1031         tp->snd_wl2 = ti->ti_ack;
1032         if (tp->snd_wnd > tp->max_sndwnd)
1033             tp->max_sndwnd = tp->snd_wnd;
1034         needoutput = 1;
1035     }

```

tcp\_input.c

## Check if send window should be updated

**1015-1023**

This if test verifies that the ACK flag is set along with any one of the three previously stated conditions. Recall that a jump was made to step6 after the receipt of a SYN in either the LISTEN or SYN\_SENT state, and in the LISTEN state the SYN does not contain an ACK.

The term *TAC* referred to in the comment is a "terminal access controller." These were Telnet clients on the ARPANET.

*1024-1027*

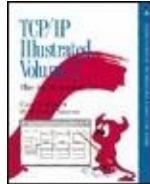
If the received segment is a pure window update (the length is 0 and the ACK does not acknowledge new data, but the advertised window is larger), the statistic `tcps_rcvwinupd` is incremented.

## Update variables

*1028-1033*

The send window is updated and new values of `snd_wl1` and `snd_wl2` are recorded. Additionally, if this advertised window is the largest one TCP has received from this peer, the new value is recorded in `max_sndwnd`. This is an attempt to guess the size of the other end's receive buffer, and it is used in [Figure 26.8](#). `needoutput` is set to 1 since the new value of `snd_wnd` might enable a segment to be sent.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.7 Urgent Mode Processing

The next part of TCP input processing handles segments with the URG flag set.

**Figure 29.16. `tcp_input` function: urgent mode processing.**

```
1035  /*
1036   * Process segments with URG.
1037   */
1038  if ((tiflags & TH_URG) && ti->ti_urp &&
1039      TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1040  /*
1041   * This is a kludge, but if we receive and accept
1042   * random urgent pointers, we'll crash in
1043   * soreceive. It's hard to imagine someone
1044   * actually wanting to send this much urgent data.
1045   */
1046  if (ti->ti_urp + so->so_rcv.sb_cc > sb_max) {
1047      ti->ti_urp = 0; /* XXX */
1048      tiflags &= ~TH_URG; /* XXX */
1049      goto dodata; /* XXX */
1050  }
```

*tcp\_input.c*

## Check if URG flag should be processed

1035-1039

These segments must have the URG flag set, a nonzero urgent offset (`ti_urp`), and the connection must not have received a FIN. The macro `TCPS_HAVERCVDFIN` is true only for the `TIME_WAIT` state, so the URG is processed in any other state. This is contrary to a comment appearing later in the code stating that the URG flag is ignored in the `CLOSE_WAIT`, `CLOSING`, `LAST_ACK`, or `TIME_WAIT` states.

## Ignore bogus urgent offsets

1040-1050

If the urgent offset plus the number of bytes already in the receive buffer exceeds the maximum size of a socket buffer, the urgent notification is ignored. The urgent offset is set to 0, the URG flag is cleared, and the rest of the urgent mode processing is skipped.

The next piece of code, shown in [Figure 29.17](#), processes the urgent pointer.

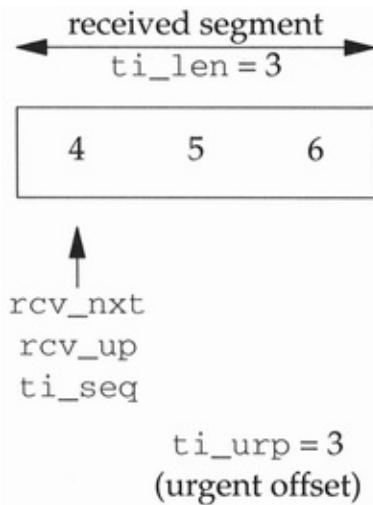
## Figure 29.17. `tcp_input` function: processing of received urgent pointer.

```
1051      /*
1052      * If this segment advances the known urgent pointer,
1053      * then mark the data stream. This should not happen
1054      * in CLOSE_WAIT, CLOSING, LAST_ACK or TIME_WAIT states since
1055      * a FIN has been received from the remote side.
1056      * In these states we ignore the URG.
1057      *
1058      * According to RFC961 (Assigned Protocols),
1059      * the urgent pointer points to the last octet
1060      * of urgent data. We continue, however,
1061      * to consider it to indicate the first octet
1062      * of data past the urgent section as the original
1063      * spec states (in one of two places).
1064      */
1065     if (SEQ_GT(ti->ti_seq + ti->ti_urp, tp->recv_up)) {
1066         tp->recv_up = ti->ti_seq + ti->ti_urp;
1067         so->so_oobmark = so->so_rcv.sb_cc +
1068             (tp->recv_up - tp->recv_nxt) - 1;
1069         if (so->so_oobmark == 0)
1070             so->so_state |= SS_RCVATMARK;
1071         sohasoutofband(so);
1072         tp->t_oobflags &= ~(TCPOOB_HAVEDATA | TCPOOB_HADDATA);
1073     }
1074     /*
1075     * Remove out-of-band data so doesn't get presented to user.
1076     * This can happen independent of advancing the URG pointer,
1077     * but if two URG's are pending at once, some out-of-band
1078     * data may creep in... ick.
1079     */
1080     if (ti->ti_urp <= ti->ti_len
1081 #ifdef SO_OOBINLINE
1082         && (so->so_options & SO_OOBINLINE) == 0
1083 #endif
1084         )
1085         tcp_pulloutofband(so, ti, m);
1086     } else {
1087         /*
1088         * If no out-of-band data is expected, pull receive
1089         * urgent pointer along with the receive window.
1090         */
1091         if (SEQ_GT(tp->recv_nxt, tp->recv_up))
1092             tp->recv_up = tp->recv_nxt;
1093     }
```

1051-1065

If the starting sequence number of the received segment plus the urgent offset exceeds the current receive urgent pointer, a new urgent pointer has been received. For example, when the 3-byte segment that was sent in [Figure 26.30](#) arrives at the receiver, we have the scenario shown in [Figure 29.18](#).

**Figure 29.18. Receiver side when segment from Figure 26.30 arrives.**



Normally the receive urgent pointer (`rcv_up`) equals `rcv_nxt`. In this example, since the if test is true (4 plus 3 is greater than 4), the new value of `rcv_up` is calculated as 7.

## Calculate receive urgent pointer

1066-1070

The out-of-band mark in the socket's receive buffer is calculated, taking into account any data bytes already in the receive buffer (`so_rcv.sb_cc`). In our example, assuming there is no data already in the receive buffer, `so_oobmark` is set to 2: that is, the byte with the sequence number 6 is considered the out-of-band byte. If this out-of-band mark is 0, the socket is currently at the out-of-band mark. This happens if the send system call that sends the out-of-band byte specifies a length of 1, and if the receive buffer is empty when this segment arrives at the other end. This reiterates that Berkeley-derived systems consider the urgent pointer to point to the first byte of data *after* the out-of-band byte.

## Notify process of TCP's urgent mode

1071-1072

`sohasoutofband` notifies the process that out-of-band data has arrived for the socket. The two flags `TCPOOB_HAVEDATA` and `TCPOOB_HADDATA` are cleared. These two flags are used with the `PRU_RCVOOB` request in [Figure 30.8](#).

## Pull out-of-band byte out of normal data stream

`1074-1085`

If the urgent offset is less than or equal to the number of bytes in the received segment, the out-of-band byte is contained in the segment. With TCP's urgent mode it is possible for the urgent offset to point to a data byte that has not yet been received. If the `SO_OOBINLINE` constant is defined (which it always is for Net/3), and if the corresponding socket option is not enabled, the receiving process wants the out-of-band byte pulled out of the normal stream of data and placed into the variable `t_iobc`. This is done by `tcp_pulloftoband`, which we cover in the next section.

Notice that the receiving process is notified that the sender has entered urgent mode, regardless of whether the byte pointed to by the urgent pointer is readable or not. This is a feature of TCP's urgent mode.

## Adjust receive urgent pointer if not urgent mode

1086-1093

When the receiver is not processing an urgent pointer, if `rcv_nxt` is greater than the receive urgent pointer, `rcv_up` is moved to the right and set equal to `rcv_nxt`. This keeps the receive urgent pointer at the left edge of the receive window so that the comparison using `SEQ_GT` at the beginning of [Figure 29.17](#) will work correctly when an URG flag is received.

If the solution to [Exercise 26.6](#) is implemented, corresponding changes will have to go into [Figures 29.16](#) and [29.17](#) also.

---

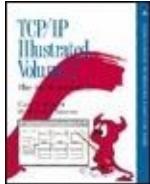
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.8 tcp\_pullofband Function

This function is called from [Figure 29.17](#) when

**1. urgent mode notification arrives in a received segment, and**

- the out-of-band byte is contained within the segment (i.e., the urgent pointer points into the received segment), and
- the SO\_OOBINLINE socket option is not enabled for this socket.

This function removes the out-of-band byte from the normal stream of data (i.e., the mbuf chain containing the received segment) and places it into the t\_iobc

variable in the TCP control block for the connection. The process reads this variable using the MSG\_OOB flag with the recv system call: the PRU\_RCVOOB request in [Figure 30.8](#). [Figure 29.19](#) shows the function.

### Figure 29.19. `tcp_pulloutofband` function: place out-of-band byte into `t_iobc`.

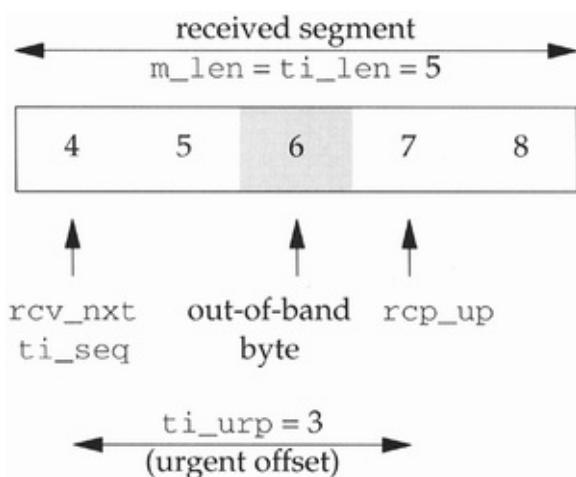
```
1282 void                                         —tcp_input.c
1283 tcp_pulloutofband(so, ti, m)
1284 struct socket *so;
1285 struct tcphdr *ti;
1286 struct mbuf *m;
1287 {
1288     int     cnt = ti->ti_urp - 1;
1289     while (cnt >= 0) {
1290         if (m->m_len > cnt) {
1291             char   *cp = mtod(m, caddr_t) + cnt;
1292             struct tcpcb *tp = sototcpcb(so);
1293             tp->t_iobc = *cp;
1294             tp->t_oobflags |= TCPOOB_HAVEDATA;
1295             bcopy(cp + 1, cp, (unsigned) (m->m_len - cnt - 1));
1296             m->m_len--;
1297             return;
1298         }
1299         cnt -= m->m_len;
1300         m = m->m_next;
1301         if (m == 0)
1302             break;
1303     }
1304     panic("tcp_pulloutofband");
1305 }
```

1282-1289

Consider the example in [Figure 29.20](#). The urgent offset is 3, therefore the urgent

pointer is 7, and the sequence number of the out-of-band byte is 6. There are 5 bytes in the received segment, all contained in a single mbuf.

**Figure 29.20. Received segment with an out-of-band byte.**



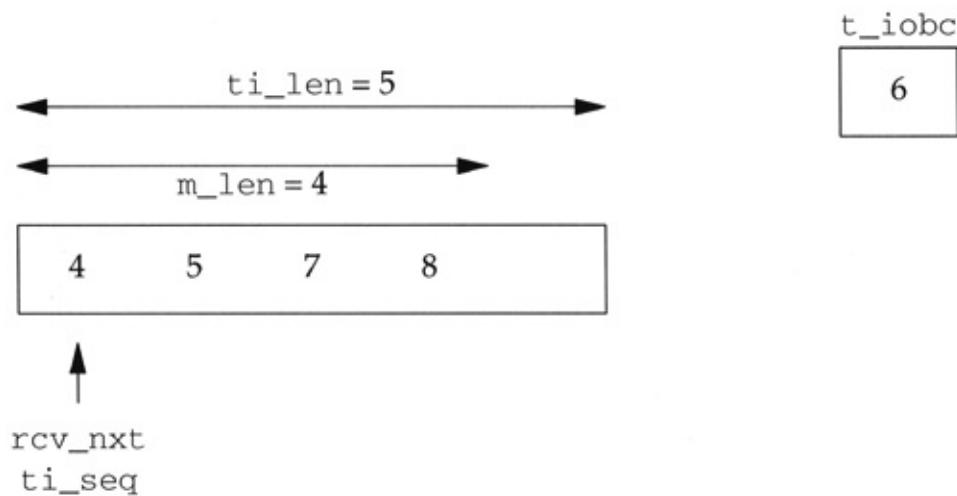
The variable `cnt` is 2 and since `m_len` (which is 5) is greater than 2, the true portion of the if statement is executed.

1290-1298

`cp` points to the shaded byte with a sequence number of 6. This is placed into the variable `t_iobc`, which contains the out-of-band byte. The `TCPOOB_HAVEDATA`

flag is set and bcopy moves the next 2 bytes (with sequence numbers 7 and 8) left 1 byte, giving the arrangement shown in [Figure 29.21](#).

**Figure 29.21. Result from Figure 29.20 after removal of out-of-band byte.**



Remember that the numbers 7 and 8 specify the sequence numbers of the data bytes, not the contents of the data bytes. The length of the mbuf is decremented from 5 to 4 but `ti_len` is left as 5, for sequencing of the segment into the socket's receive buffer. Both the `TCP_REASS` macro and the `tcp_reass` function (which are called in the next

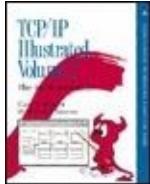
section) increment `recv_nxt` by `ti_len`, which in this example must be 5, because the next expected receive sequence number is 9. Also notice in this function that the length field in the packet header (`m_pkthdr.len`) in the first mbuf is not decremented by 1. This is because that length field is not used by `sbappend`, which appends the data to the socket's receive buffer.

## Skip to next mbuf in chain

**1299-1302**

The out-of-band byte is not contained in this mbuf, so `cnt` is decremented by the number of bytes in the mbuf and the next mbuf in the chain is processed. Since this function is called only when the urgent offset points into the received segment, if there is not another mbuf on the chain, the break causes the call to panic.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.9 Processing of Received Data

`tcp_input` continues by taking the received data (if any) and either appending it to the socket's receive buffer (if it is the next expected segment) or placing it onto the socket's out-of-order queue. [Figure 29.22](#) shows the code that performs this task.

**Figure 29.22. `tcp_input` function: merge received data into sequencing queue for socket.**

```

1094     dodata:           /* XXX */
1095     /*
1096      * Process the segment text, merging it into the TCP sequencing queue,
1097      * and arranging for acknowledgment of receipt if necessary.
1098      * This process logically involves adjusting tp->rcv_wnd as data
1099      * is presented to the user (this happens in tcp_usrreq.c,
1100      * case PRU_RCVD). If a FIN has already been received on this
1101      * connection then we just ignore the text.
1102      */
1103     if ((ti->ti_len || (tiflags & TH_FIN)) &&
1104         TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1105         TCP_REASS(tp, ti, m, so, tiflags);
1106         /*
1107          * Note the amount of data that peer has sent into
1108          * our window, in order to estimate the sender's
1109          * buffer size.
1110          */
1111         len = so->so_rcv.sb_hiwat - (tp->rcv_adv - tp->rcv_nxt);
1112     } else {
1113         m_freem(m);
1114         tiflags &= ~TH_FIN;
1115     }

```

tcp\_input.c

## 1094-1105

Segment data is processed if

- 1. the length of the received data is greater than 0 or the FIN flag is set, and**

- a FIN has not yet been received for the connection.

The macro TCP\_REASS processes the data. If the data is in sequence (i.e., the next expected data for this connection), the delayed-ACK flag is set, `rcv_nxt` is incremented, and the data is appended to the socket's receive buffer. If the data is out of order, the macro calls `tcp_reass` to

add the data to the connection's reassembly queue (which might fill a hole and cause already-queued data to be appended to the socket's receive buffer).

Recall that the final argument to the macro (tiflags) can be modified. Specifically, if the data is out of order, `tcp_reass` sets tiflags to 0, clearing the FIN flag (if it was set). That's why the if statement is true if the FIN flag is set even if there is no data in the segment.

Consider the following example. A connection is established and the sender immediately transmits three segments: one with bytes 11024, another with bytes 10252048, and another with the FIN flag but no data. The first segment is lost, so when the second arrives (bytes 10252048) the receiver places it onto the out-of-order list and generates an immediate ACK. When the third segment with the FIN flag is received, the code in [Figure 29.22](#) is executed. Even though the data length is 0, since the FIN flag is set, `TCP_REASS` is invoked, which calls `tcp_reass`. Since `ti_seq` (2049, the sequence number of the

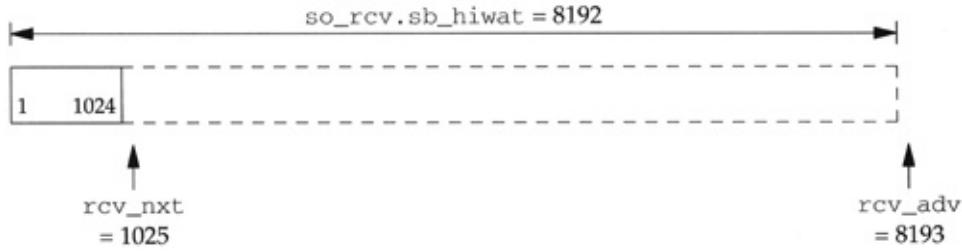
FIN) does not equal rcv\_nxt (1), tcp\_reass returns 0 ([Figure 27.23](#)), which in the TCP\_REASS macro sets tiflags to 0. This clears the FIN flag, preventing the code that follows ([Section 29.10](#)) from processing the FIN flag.

## Guess size of other end's send buffer

*1106-1111*

The calculation of len is attempt to guess the size of the other end's send buffer. Consider the following example. A socket has a receive buffer size of 8192 (the Net/3 default), so TCP advertises a window of 8192 in its SYN. The first segment with bytes 11024 is then received. [Figure 29.23](#) shows the state of the receive space after TCP\_REASS has incremented rcv\_nxt to account for the received segment.

**Figure 29.23. Receipt of bytes 11024 into a 8192-byte receive window.**



The calculation of len yields 1024. The value of len will increase as the other end sends more data into the receive window, but it will never exceed the size of the other end's send buffer. Recall that the variable max\_sndwnd, calculated in [Figure 29.15](#), is an attempt to guess the size of the other end's receive buffer.

This variable len is never used! It is left over code from Net/1 when the variable max\_rcvd was stored in the TCP control block after the calculation of len:

```
if (len > tp->max_rcvd)
    tp->max_rcvd = len;
```

But even in Net/1 the variable max\_rcvd was never used.

*1112-1115*

If the length is 0 and the FIN flag is not

set, or if a FIN has already been received for the connection, the received mbuf chain is discarded and the FIN flag is cleared.

---

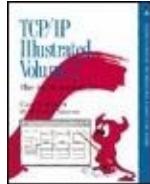
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.10 FIN Processing

The next step in `tcp_input`, shown in Figure 29.24, handles the FIN flag.

**Figure 29.24. `tcp_input` function: FIN processing, first half.**

```
1116  /*
1117   * If FIN is received ACK the FIN and let the user know
1118   * that the connection is closing.
1119   */
1120  if (tiflags & TH_FIN) {
1121      if (TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1122          socantrcvmore(so);
1123          tp->t_flags |= TF_ACKNOW;
1124          tp->rcv_nxt++;
1125      }
1126      switch (tp->t_state) {
1127          /*
1128           * In SYN_RECEIVED and ESTABLISHED states
1129           * enter the CLOSE_WAIT state.
1130           */
1131      case TCPS_SYN_RECEIVED:
1132      case TCPS_ESTABLISHED:
1133          tp->t_state = TCPS_CLOSE_WAIT;
1134          break;
```

*tcp\_input.c*

## Process first FIN received on connection

### 1116-1125

If the FIN flag is set and this is the first FIN received for this connection, socantrcvmore marks the socket as write-only, TF\_ACKNOW is set to acknowledge the FIN immediately (i.e., it is not delayed), and rcv\_nxt steps over the FIN in the sequence space.

### 1126

The remainder of FIN processing is handled by a switch that depends on the connection state. Notice that the FIN is not processed in the CLOSED, LISTEN, or SYN\_SENT states, since in these three states a SYN has not been received to synchronize the received sequence number, making it impossible to validate the sequence number of the FIN. A FIN is also ignored in the CLOSING, CLOSE\_WAIT, and LAST\_ACK states, because in these three states the FIN is a duplicate.

## **SYN\_RCVD or ESTABLISHED states**

*1127-1134*

From either the ESTABLISHED or SYN\_RCVD states, the CLOSE\_WAIT state is entered.

The receipt of a FIN in the SYN\_RCVD state is unusual, but legal. It is not shown in [Figure 24.15](#). It means a socket is in the LISTEN state when a segment containing a SYN and a FIN is received. Alternatively, a SYN is received for a listening socket, moving the connection to the SYN\_RCVD state but before the ACK is received a FIN is received. (We know the segsment does not contain a valid ACK, because if it did the code in [Figure 29.2](#) would have moved the connection to the ESTABLISHED state.)

The next part of FIN processing is shown in [Figure 29.25](#)

**[Figure 29.25. tcp\\_input function: FIN](#)**

## processing, second half.

```
1135      /*
1136      * If still in FIN_WAIT_1 state FIN has not been acked so
1137      * enter the CLOSING state.
1138      */
1139     case TCPS_FIN_WAIT_1:
1140         tp->t_state = TCPS_CLOSING;
1141         break;
1142     /*
1143     * In FIN_WAIT_2 state enter the TIME_WAIT state,
1144     * starting the time-wait timer, turning off the other
1145     * standard timers.
1146     */
1147     case TCPS_FIN_WAIT_2:
1148         tp->t_state = TCPS_TIME_WAIT;
1149         tcp_canceltimers(tp);
1150         tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1151         soisdisconnected(so);
1152         break;
1153     /*
1154     * In TIME_WAIT state restart the 2 MSL time_wait timer.
1155     */
1156     case TCPS_TIME_WAIT:
1157         tp->t_timer[TCPT_2MSL] = 2 * TCPTV_MSL;
1158         break;
1159     }
1160 }
```

## FIN\_WAIT\_1 state

1135-1141

Since ACK processing is already complete for this segment, if the connection is in the FIN\_WAIT\_1 state when the FIN is processed, it means a simultaneous close is taking place the two FINs from each end have passed in the network. The connection enters the CLOSING state.

## **FIN\_WAIT\_2 state**

*1142-1148*

The receipt of the FIN moves the connection into the TIME\_WAIT state. When a segment containing a FIN and an ACK is received in the FIN\_WAIT\_1 state (the typical scenario), although [Figure 24.15](#) shows the transition directly from the FIN\_WAIT\_1 state to the TIME\_WAIT state, the ACK is processed in [Figure 29.11](#), moving the connection to the FIN\_WAIT\_2 state. The FIN processing here moves the connection into the TIME\_WAIT state. Because the ACK is processed before the FIN, the FIN\_WAIT\_2 state is always passed through, albeit momentarily.

## **Start TIME\_WAIT timer**

*1149-1152*

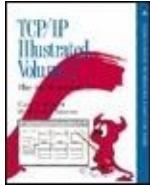
Any pending TCP timer is turned off and the TIME\_WAIT timer is started with a value of twice the MSL. (If the received

segment contained a FIN and an ACK, [Figure 29.11](#) started the FIN\_WAIT\_2 timer.) The socket is disconnected.

## TIME\_WAIT state

### 1153-1159

If a FIN arrives in the TIME\_WAIT state, it is a duplicate, and similar to [Figure 29.14](#), the TIME\_WAIT timer is restarted with a value of twice the MSL.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.11 Final Processing

The final part of the slow path through `tcp_input` along with the label `dropafterack` is shown in [Figure 29.26](#).

**Figure 29.26. `tcp_input` function: final processing.**

```

1161     if (so->so_options & SO_DEBUG)
1162         tcp_trace(TA_INPUT, ostate, tp, &tcp_saveti, 0);
1163     /*
1164      * Return any desired output.
1165      */
1166     if (needoutput || (tp->t_flags & TF_ACKNOW))
1167         (void) tcp_output(tp);
1168     return;
1169 dropafterack:
1170     /*
1171      * Generate an ACK dropping incoming segment if it occupies
1172      * sequence space, where the ACK reflects our state.
1173      */
1174     if (tiflags & TH_RST)
1175         goto drop;
1176     m_freem(m);
1177     tp->t_flags |= TF_ACKNOW;
1178     (void) tcp_output(tp);
1179     return;

```

*tcp\_input.c*

## SO\_DEBUG socket option

*1161-1162*

If the SO\_DEBUG socket option is enabled, `tcp_trace` appends the trace record to the kernel's circular buffer. Remember that the code in [Figure 28.7](#) saved both the original connection state and the IP and TCP headers, since these values may have changed in this function.

## Call `tcp_output`

*1163-1168*

If either the needoutput flag was set

([Figures 29.6](#) and [29.15](#)) or if an immediate ACK is required, `tcp_output` is called.

## dropafterack

[1169-1179](#)

An ACK is generated only if the RST flag was not set. (A segment with an RST is never ACKed.) The mbuf chain containing the received segment is released, and `tcp_output` generates an immediate ACK.

[Figure 29.27](#) completes the `tcp_input` function.

**Figure 29.27. `tcp_input` function: final processing.**

---

```

1180     dropwithreset:
1181     /*
1182      * Generate an RST, dropping incoming segment.
1183      * Make ACK acceptable to originator of segment.
1184      * Don't bother to respond if destination was broadcast/multicast.
1185      */
1186     if ((tiflags & TH_RST) || m->m_flags & (M_BCAST | M_MCAST) ||
1187         IN_MULTICAST(ti->ti_dst.s_addr))
1188     goto drop;
1189     if (tiflags & TH_ACK)
1190     tcp_respond(tp, ti, m, (tcp_seq) 0, ti->ti_ack, TH_RST);
1191     else {
1192       if (tiflags & TH_SYN)
1193         ti->ti_len++;
1194       tcp_respond(tp, ti, m, ti->ti_seq + ti->ti_len, (tcp_seq) 0,
1195                   TH_RST | TH_ACK);
1196     }
1197     /* destroy temporarily created socket */
1198     if (dropsocket)
1199       (void) soabort(so);
1200     return;
1201 drop:
1202   /*
1203    * Drop space held by incoming segment and return.
1204    */
1205   if (tp && (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
1206     tcp_trace(TA_DROP, ostate, tp, &tcp_saveti, 0);
1207   m_freem(m);
1208   /* destroy temporarily created socket */
1209   if (dropsocket)
1210     (void) soabort(so);
1211   return;
1212 }
```

---

tcp\_input.c

## dropwithreset

*1180-1188*

An RST is generated unless the received segment also contained an RST, or the received segment was sent as a broadcast or multicast. An RST is never generated in response to an RST, since this could lead to RST storms (a continual exchange of RST segments between two end points).

This code contains the same error that

we noted in [Figure 28.16](#): it does not check whether the destination address of the received segment was a broadcast address.

Similarly, the destination address argument to `IN_MULTICAST` needs to be converted to host byte order.

## **Sequence number and acknowledgment number of RST segment**

**1189-1196**

The values of the sequence number field, the acknowledgment field, and the ACK flag of the RST segment depend on whether the received segment contained an ACK.

[Figure 29.28](#) summarizes these fields in the RST segment that is generated.

**Figure 29.28. Values of fields in RST segment generated.**

received segment	RST segment generated		
	seq#	ack. field	flags
contains ACK ACK-less	received ack. field 0	0 received seq# field	TH_RST TH_RST   TH_ACK

Realize that the ACK flag is normally set in all segments except when an initial SYN is sent ([Figure 24.16](#)). The fourth argument to `tcp_respond` is the acknowledgment field, and the fifth argument is the sequence number.

## Rejecting connections

*1192-1193*

If the SYN flag is set, `ti_len` must be incremented by 1, causing the acknowledgment field of the RST to be 1 greater than the received sequence number of the SYN. This code is executed when a SYN arrives for a nonexistent server. When the Internet PCB is not found in [Figure 28.6](#), a jump is made to `dropwithreset`. But for the received RST to be acceptable to the other end, the acknowledgment field must ACK the SYN ([Figure 28.18](#)). Figure 18.14 of Volume 1 contains an example of this type of RST

segment.

Finally note that `tcp_respond` builds the RST in the first mbuf of the received chain and releases any remaining mbufs in the chain. When that mbuf finally makes its way to the device driver, it will be discarded.

## Destroy temporarily created socket

*1197-1199*

If a temporary socket was created in [Figure 28.7](#) for a listening server, but the code in [Figure 28.16](#) found the received segment to contain an error, `dropsocket` will be 1. If so, that socket is now destroyed.

## Drop (without ACK or RST)

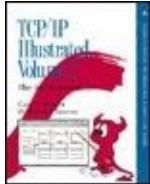
*1201-1206*

`tcp_trace` is called when a segment is dropped without generating an ACK or an RST. If the `SO_DEBUG` flag is set and an

ACK is generated, `tcp_output` generates a trace record. If the `SO_DEBUG` flag is set and an RST is generated, a trace record is not generated for the RST.

## *1207-1211*

The mbuf chain containing the received segment is released and the temporary socket is destroyed if `dropsocket` is nonzero.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

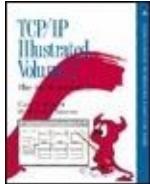
## Chapter 29. TCP Input (Continued)

### 29.12 Implementation Refinements

The refinements to speed up TCP processing are similar to the ones described for UDP ([Section 23.12](#)). Multiple passes over the data should be avoided and the checksum computation should be combined with a copy. [[Dalton et al. 1993](#)] describe these modifications.

The linear search of the TCP PCBs is also a bottleneck when the number of connections increases. [[McKenney and Dove 1992](#)] address this problem by replacing the linear search with hash tables.

[[Partridge 1993](#)] describes a research implementation being developed by Van Jacobson that greatly reduces the TCP input processing. The received packet is processed by IP (about 25 instructions on a RISC system), then by a demultiplexer to locate the PCB (about 10 instructions), and then by TCP (about 30 instructions). These 30 instructions perform header prediction and calculate the pseudo-header checksum. If the segment passes the header prediction test, contains data, and the process is waiting for the data, the data is copied into the process buffer and the remainder of the TCP checksum is calculated and verified (a one-pass copy and checksum). If the TCP header prediction fails, the slow path through the TCP input processing occurs.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.13 Header Compression

We now describe TCP *header compression*. Although header compression is not part of TCP input, we needed to cover TCP thoroughly before describing header compression. Header compression is described in detail in RFC 1144 [[Jacobson 1990a](#)]. It was designed by Van Jacobson and is sometimes called *VJ header compression*. Our purpose in this section is not to go through the header compression source code (a well-commented version of which is presented in RFC 1144, and which is approximately the same size as `tcp_output`), but to provide an overview of the algorithm. Be sure to distinguish between header prediction ([Section 28.4](#))

and header compression.

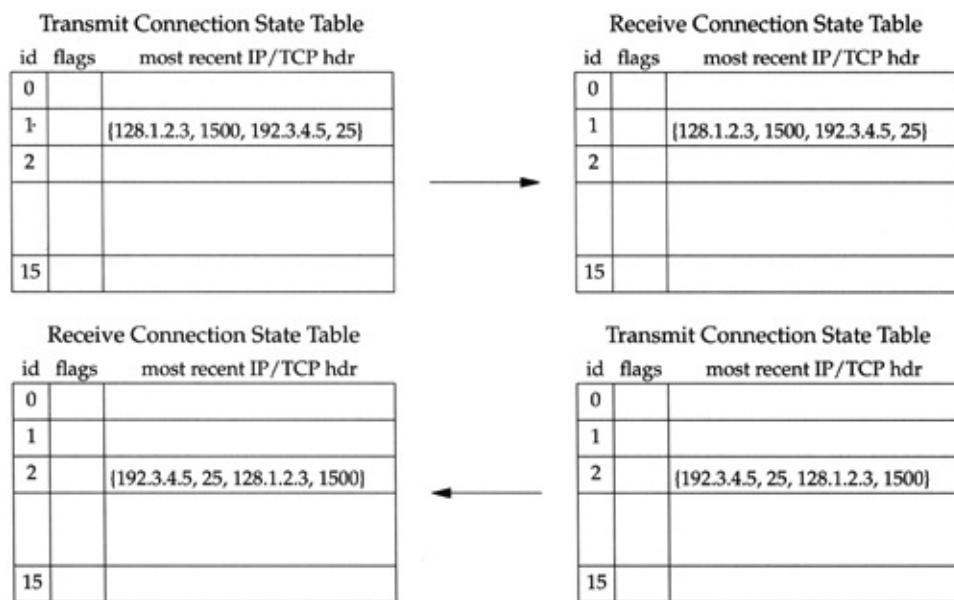
## Introduction

Most implementations of SLIP and PPP support header compression. Although header compression could, in theory, be used with any data link, it is intended for slow-speed serial links. Header compression works with TCP segments only if it does nothing with other IP datagrams (e.g., ICMP, IGMP, UDP, etc.). Header compression reduces the size of the combined IP/TCP header from its normal 40 bytes to as few as 3 bytes. This reduces the size of a typical TCP segment from an interactive application such as Rlogin or Telnet from 41 bytes to 4 bytes a big saving on a slow-speed serial link.

Each end of the serial link maintains two connection state tables, one for datagrams sent and one for datagrams received. Each table allows a maximum of 256 entries, but typically there are 16 entries in this table, allowing up to 16 different TCP connections to be compressed at any time.

Each entry contains an 8-bit connection ID (hence the limit of 256), some flags, and the complete uncompressed IP/TCP header from the most recent datagram. The 96-bit socket pair that uniquely identifies each connection—the source and destination IP addresses and source and destination TCP ports—are contained in this uncompressed header. [Figure 29.29](#) shows an example of these tables.

**Figure 29.29. A pair of connection state tables at each end of a link (e.g., SLIP link).**



Since a TCP connection is full duplex,

header compression can be applied in both directions. Each end must implement both compression and decompression. A connection appears in both tables, as shown in [Figure 29.29](#). In this example, the entry with a connection ID of 1 in the top two tables has a source IP address of 128.1.2.3, source TCP port of 1500, destination IP address of 192.3.4.5, and a destination TCP port of 25. The entry with a connection ID of 2 in the bottom two tables is for the other direction of the same connection.

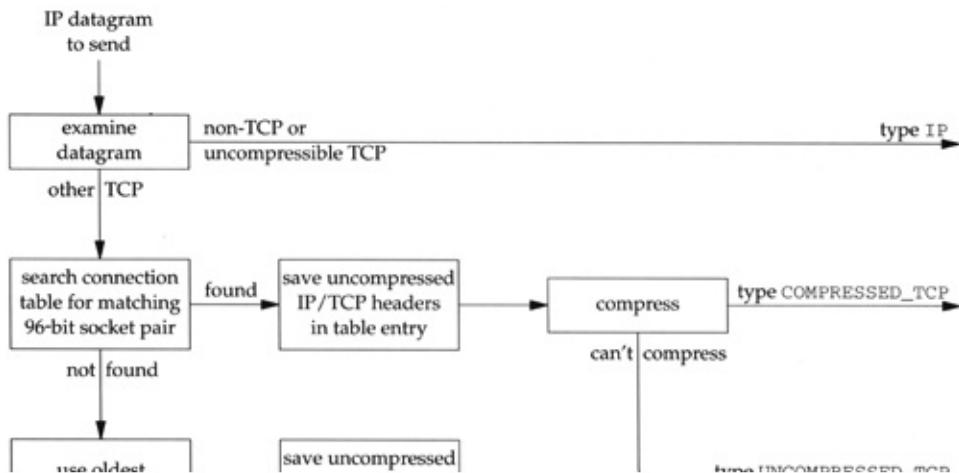
We show these tables as arrays, but the source code defines each entry as a structure, and a connection table is a circular linked list of these structures. The most recently used structure is stored at the head of the list.

By saving the most recent uncompressed header at each end, only the *differences* in various header fields from the previous datagram to the current datagram are transmitted across the link (along with a special first byte indicating which fields follow). Since some header fields don't

change at all from one datagram to the next, and other header fields change by small amounts, this differential coding provides the savings. Header compression works with the IP and TCP headers only if the data contents of the TCP segment are not modified.

[Figure 29.30](#) shows the steps involved at the sending side when it has an IP datagram to send across a link using header compression.

### Figure 29.30. Steps involved in header compression at sender side.



Three different types of datagrams are

sent and must be recognized at the receiver:

- 1. Type IP is specified with the high-order 4 bits of the first byte equal to 4. This is the normal IP version number in the IP header ([Figure 8.8](#)). The normal, uncompressed datagram is transmitted across the link.**
  - Type COMPRESSED\_TCP is specified by setting the high-order bit of the first byte. This looks like an IP version between 8 and 15 (i.e., the remaining 7 bits of this byte are used by the compression algorithm). The compressed header and uncompressed data are transmitted across the link, as we describe later in this section.
  - Type UNCOMPRESSED\_TCP is specified with the high-order 4 bits of the first byte equal to 7. The normal, uncompressed datagram is transmitted across the link, but the IP protocol field (which equals 6 for TCP), is replaced with the connection ID. This identifies the connection state

table entry for the receiver.

The receiver can identify the datagram type by examining its first byte. The code that does this was shown in [Figure 5.13](#). In [Figure 5.16](#) the sender calls `sl_compress_tcp` to check if a TCP segment is compressible, and the return value of this function is logically ORed into the first byte of the datagram.

[Figure 29.31](#) shows an illustration of the first byte that is sent across the link.

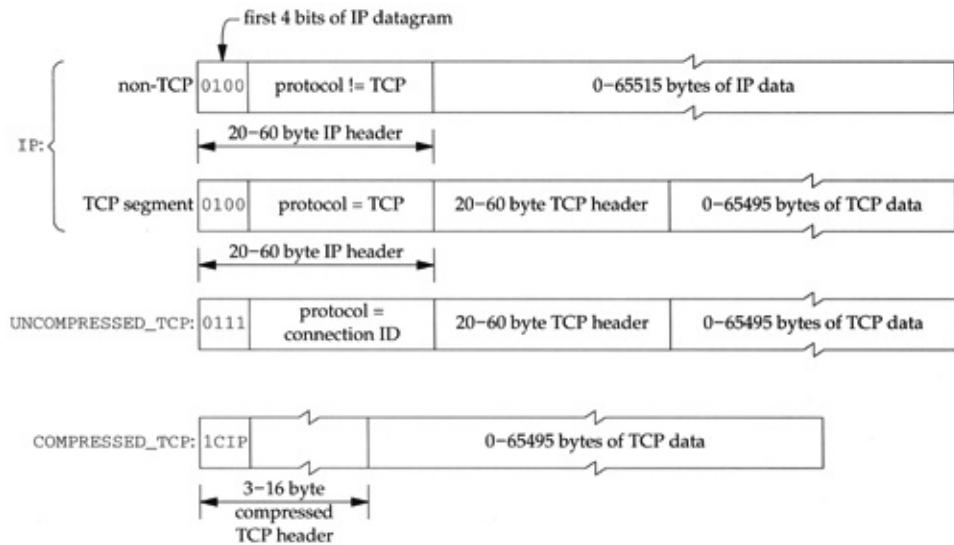
### Figure 29.31. First byte transmitted across link.

first byte transmitted across link	4-bit version		4-bit header length				IP UNCOMPRESSED_TCP COMPRESSED_TCP
	0	1	0	0	-	-	-
	0	1	1	1	-	-	-
	1	C	I	P	S	A	W

The 4 bits shown as "-" comprise the normal IP header length field. The 7 bits shown as C, I, P, S, A, W, and U indicate which optional fields follow. We describe these fields shortly.

Figure 29.32 shows the complete IP datagram for the various datagrams that are sent.

### Figure 29.32. Different types of IP datagrams possible with header compression.



We show two datagrams with a type of IP: one that is not a TCP segment (e.g., a protocol of UDP, ICMP, or IGMP), and one that is a TCP segment. This is to illustrate the differences between the TCP segment sent as type IP and the TCP segment sent as type UNCOMPRESSED\_TCP: the first 4 bits are different as is the protocol field in

the IP header.

Datagrams are not candidates for header compression if the protocol is not TCP, or if the protocol is TCP but any one of the following conditions is true.

- The datagram is an IP fragment: either the fragment offset is nonzero or the more-fragments bit is set.
- Any one of the SYN, FIN, or RST flags is set.
- The ACK flag is not set.

If any one of these three conditions is true, the datagram is sent as type IP.

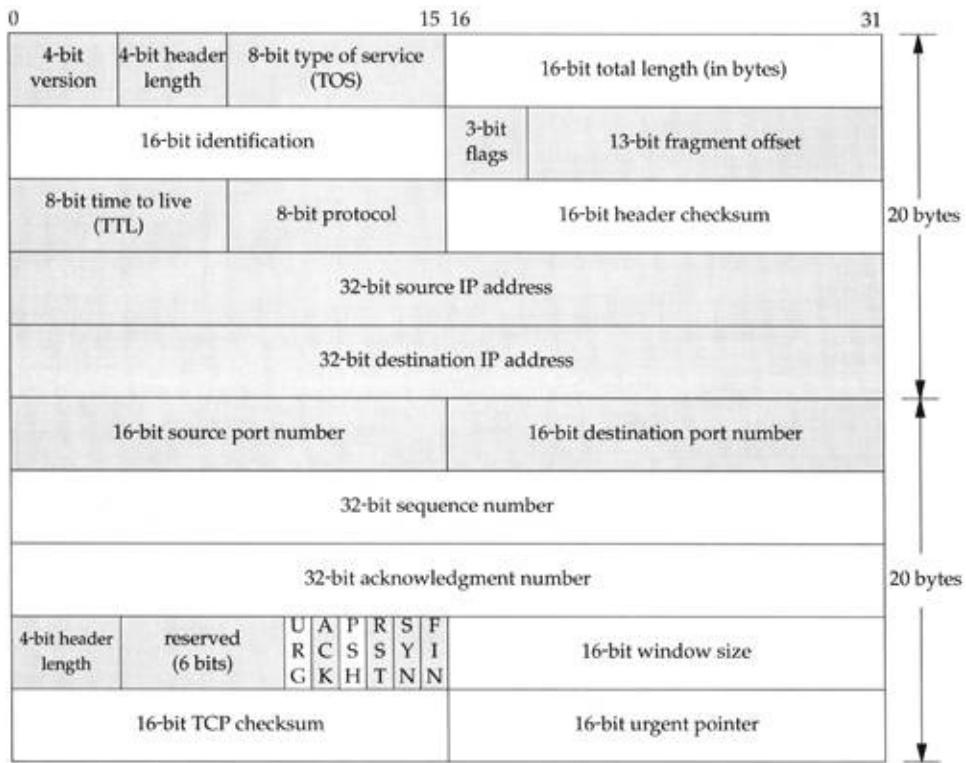
Furthermore, even if the datagram is a TCP segment that looks compressible, it is possible to abort the compression and send the datagram as type UNCOMPRESSED\_TCP if certain fields have changed between the current datagram and the last datagram sent for this connection. These are fields that normally do not change for a given connection, so the compression scheme was not designed

to encode their differences from one datagram to the next. The TOS field and the don't fragment bit are examples. Also, when the differences in some fields are greater than 65535, the compression algorithm fails and the datagram is sent uncompressed.

## Compression of Header Fields

We now describe how the fields in the IP and TCP headers, shown in [Figure 29.33](#), are compressed. The shaded fields normally don't change during a connection.

**Figure 29.33. Combined IP and TCP headers: shaded fields normally don't change.**

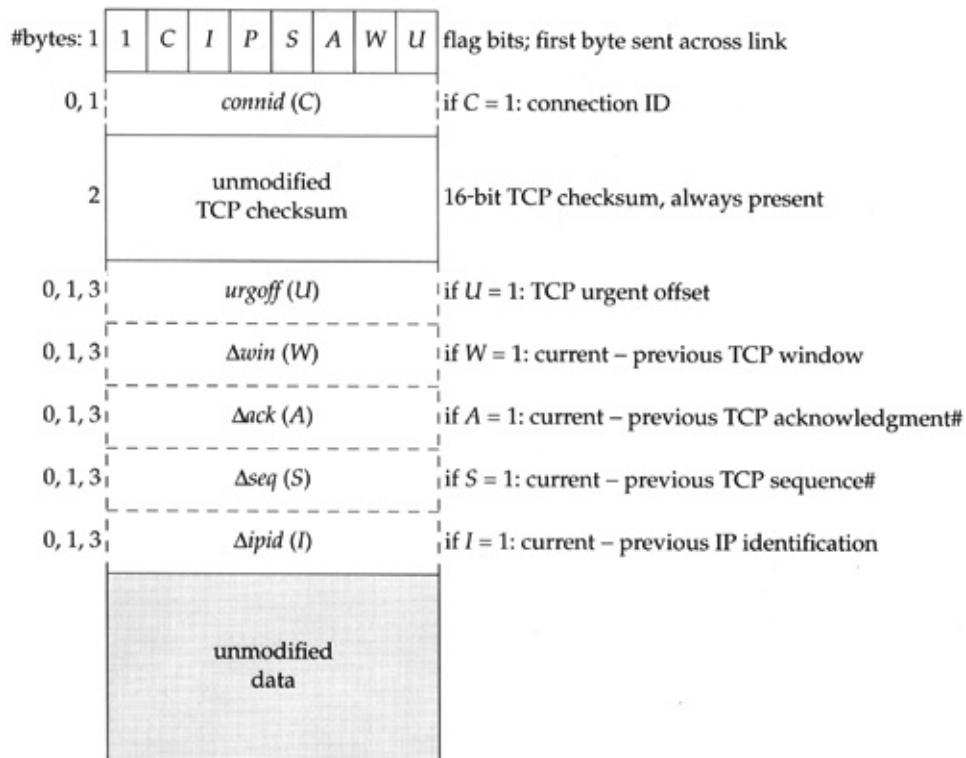


If any of the shaded fields have changed from the previous segment on this connection to the current segment, the segment is sent uncompressed. We don't show IP options or TCP options in this figure, but if either are present and have changed from the previous segment, the segment is sent uncompressed ([Exercise 29.7](#)).

If the algorithm transmitted only the nonshaded fields when the shaded fields do not change from the previous segment, about a 50% savings would result. VJ header compression does even better than

this, by knowing which fields in the IP and TCP headers *normally* don't change. [Figure 29.34](#) shows the format of the compressed IP/TCP header.

**Figure 29.34. Format of compressed IP/TCP header.**



The smallest compressed header consists of 3 bytes: the first byte (the flag bits) followed by the 16-bit TCP checksum. For protection against possible link errors, the TCP checksum is always transmitted

without any change. (SLIP provides no link-layer checksum, although PPP does provide one.)

The other six fields, *connid*, *urgoff*, *Dwin*, *Dack*, *Dseq*, and *Dipid*, are optional. We show the number of bytes used to encode all the fields to the left of the field in [Figure 29.34](#). The largest compressed header appears to be 19 bytes, but we'll see shortly that the 4 bits *SAWU* can never be set at the same time in a compressed header, so the largest size is actually 16 bytes.

Six of the 7 bits in the first byte specify which of the six optional fields are present. The high-order bit of the first byte is always set to 1. This identifies the datagram type as COMPRESSED\_TCP. [Figure 29.35](#) summarizes the 7 bits, which we now describe.

**Figure 29.35. The 7 bits in the compressed header.**

Flag bit	Description	Structure member	Meaning if flag = 0	Meaning if flag = 1
C	connection ID		same connection ID as last	<i>connid</i> = connection ID
I	IP identification		<i>ip_id</i> has increased by 1	$\Delta ipid$ = current – previous
P	TCP push flag		PSH flag off	PSH flag on
S	TCP sequence#		same <i>th_seq</i> as last	$\Delta seq$ = current – previous
A	TCP acknowledgment#		same <i>th_ack</i> as last	$\Delta ack$ = current – previous
W	TCP window		same <i>th_win</i> as last	$\Delta win$ = current – previous
U	TCP urgent offset		URG flag not set	<i>urgoff</i> = urgent offset

- C If this bit is 0, this segment has the same connection ID as the previous compressed or uncompressed segment. If this flag is 1, *connid* is the connection ID, a value between 0 and 255.
- I If this bit is 0, the IP identification field has increased by 1 (the typical case). If this bit is 1, *Dipid* is the current value of *ip\_id* minus its previous value.
- P This bit is a copy of the PSH flag from the TCP segment. Since the PSH flag doesn't follow any established pattern, it must be explicitly specified for each segment.
- S If this bit is 0, the TCP sequence number has not changed. If this bit is 1,

*Dseq* is the current value of *th\_seq* minus its previous value.

*A* If this bit is 0, the TCP acknowledgment number has not changed (the typical case). If this bit is 1, *Dack* is the current value of *th\_ack* minus its previous value.

*W*If this bit is 0, the TCP window has not changed (the typical case). If this bit is 1, *Dwin* is the current value of *th\_win* minus its previous value.

*U* If this bit is 0, the URG flag in the segment is not set and the urgent offset has not changed from its previous value (the typical case). If this bit is 1, *urgoff* is the current value of *th\_urg* and the URG flag is set. If the urgent offset changes without the URG flag being set, the segment is sent uncompressed. (This often occurs in the first segment following urgent data.)

The differences are encoded as the current value minus the previous value, because most of these differences will be small positive numbers (with *Dwin* being an exception) given the way these fields normally change.

We note that five of the optional fields in [Figure 29.34](#) are encoded in 0, 1, or 3 bytes.

- 0 If the corresponding flag is not set, bytes: nothing is encoded for the field.
- 1 If the value to send is between 1 byte: and 255, a single byte encodes the value.
- 3 If the value to send is either 0 or bytes: between 256 and 65535, 3 bytes encode the value: the first byte is 0, followed by the 2-byte value.  
This always works for the three 16-bit values, *urgoff*, *Dwin*, and *Dipid*; but if the difference to encode for

the two 32-bit values, Dack and Dseq, is less than 0 or greater than 65535, the segment is sent uncompressed.

If we compare the nonshaded fields in [Figure 29.33](#) with the possible fields in [Figure 29.34](#) we notice that some fields are never transmitted.

- The IP total length field is not transmitted since most link layers provide the length of a received message to the receiver.
- Since the only field in the IP header that is being transmitted is the identification field, the IP checksum is also omitted. This is a hop-by-hop checksum that protects only the IP header across any given link.

## Special Cases

Two common cases are detected and transmitted as special combinations of the

4 low-order bits: *SAWU*. Since urgent data is rare, if the URG flag in the segment is set and both the sequence number and window also change (implying that the 4 low-order bits would be 1011 or 1111), the segment is sent uncompressed.

Therefore if the 4 low-order bits are sent as 1011 (called \*SA) or 1111 (called \*S), the following two special cases apply:

\*SA The sequence number and acknowledgment number both increase by the amount of data in the last segment, the window and urgent offset don't change, and the URG flag is not set. This special case avoids encoding both Dseq and Dack.

This case occurs frequently for both directions of echoed terminal traffic. Figures 19.3 and 19.4 of Volume 1 give examples of this type of data flow across an Rlogin connection.

\*S The sequence number changes by the amount of data in the last segment,

the acknowledgment number, window, and urgent offset don't change, and the URG flag is not set. This special case avoids encoding Dseq.

This case occurs frequently for the sending side of a unidirectional data transfer (e.g., FTP). Figures 20.1, 20.2, and 20.3 of Volume 1 give examples of this type of data transfer. This case also occurs for the sender of nonechoed terminal traffic (e.g., commands that are not echoed by a full-screen editor).

## Examples

Two simple examples were run across the SLIP link between the systems bsdi and slip in [Figure 1.17](#). This SLIP link uses header compression in both directions. The tcpdump program described in Appendix A of Volume 1 was also run on the host bsdi to save a copy of all the frames. This program has an option that outputs the compressed header, showing all the fields

in Figure 29.34.

Two traces were obtained: a short portion of an Rlogin connection and a file transfer from bsdi to slip using FTP. Figure 29.36 shows a summary of the different frame types for both connections.

**Figure 29.36. Counts of different frame types for Rlogin and FTP connections.**

frame type	Rlogin		FTP	
	input	output	input	output
IP	1	1	5	5
UNCOMPRESSED_TCP	3	2	2	3
COMPRESSED_TCP				
*SA special case	75	75	0	0
*S special case	25	1	1	325
nonspecial	9	93	337	13
Total	113	172	345	346

The two entries of 75 verify our claim that this special case often occurs for both directions of echoed terminal traffic. The entry of 325 verifies our claim that this special case occurs frequently for the sending side of a unidirectional data transfer.

The 10 frames of type IP for the FTP

example correspond to four segments with the SYN flag set and six segments with the FIN flag set. FTP uses two connections: one for the interactive commands and one for the file transfer.

The UNCOMPRESSED\_TCP frame types normally correspond to the first segment following connection establishment, the one that establishes the connection ID. An additional few are seen in these examples when the type of service is set (the Net/3 Rlogin and FTP clients and servers all set the TOS field *after* the connection is established).

[Figure 29.37](#) shows the distribution of the compressed-header sizes. The average size of the compressed header for the final four columns in [Figure 29.37](#) is 3.1, 4.1, 6.0, and 3.3 bytes, a significant savings compared to the uncompressed 40-byte headers, especially for the interactive connection.

### **Figure 29.37. Distribution of compressed-header sizes.**

#bytes	Rlogin		FTP	
	input	output	input	output
3	102	44	2	250
4		94		78
5	7	12	5	2
6		6	325	5
7		13	2	1
8				1
9			4	1
Total	109	169	338	338

Almost all of the 325 6-byte headers in the FTP input column contain only a *Dack* of 256, which being greater than 255 is encoded in 3 bytes. The SLIP MTU is 296, so TCP uses an MSS of 256. Almost all of the 250 3-byte headers in the FTP output column contain the \*S special case (sequence number change only) with a change of 256 bytes. But since this change refers to the amount of data in the previous segment, nothing is transmitted other than the flag byte and the TCP checksum. The 78 4-byte headers in the FTP output column are this same special case, but with a change in the IP identification field also ([Exercise 29.8](#)).

## Configuration

Header compression must be enabled on a

given SLIP or PPP link. With a SLIP link there are normally two flags that can be set when the interface is configured: enable header compression and autoenable header compression. These two flags are set using the link0 and link2 flags to the ifconfig command, respectively. Normally a client (the dialin host) decides whether to use header compression or not. The server (the host or terminal server to which the client dials in) specifies the autoenable flag only. If header compression is enabled by the client, its TCP will send a datagram of type UNCOMPRESSED\_TCP to specify the connection ID. When the server sees this packet it enables header compression (since it was in the autoenable mode). If the server never sees this type of packet, it never enables header compression for this line.

PPP allows the negotiation of options between the two ends of the link when the link is established. One of the options that can be negotiated is whether to use header compression or not.

---

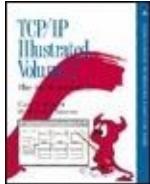
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 29. TCP Input (Continued)

### 29.14 Summary

This chapter completes our detailed look at TCP input processing. We started with the processing of an ACK in the SYN\_RCVD state, which completes a passive open, a simultaneous open, or a self-connect.

The fast retransmit algorithm lets TCP detect a dropped segment after receiving a specified number of consecutive duplicate ACKs and retransmit the segment before the retransmission timer expires. Net/3 combines the fast retransmit algorithm with the fast recovery algorithm, which tries to keep the data flowing from the sender to the receiver, albeit at a slower rate, using congestion avoidance but not

slow start.

ACK processing then discards the acknowledged data from the socket's send buffer and handles a few TCP states specially, when the receipt of an ACK changes the connection state.

The URG flag is processed, if set, and TCP's urgent mode is mapped into the socket abstraction of out-of-band data. This is complicated because the process can receive the out-of-band byte inline or in a special out-of-band buffer, and TCP can receive urgent notification before the data byte referenced by the urgent pointer has been received.

TCP input processing completes by calling `TCP_REASS` to merge the received data into either the socket's receive buffer or the socket's out-of-order queue, processing the FIN flag, and calling `tcp_output` if a segment must be generated in response to the received segment.

TCP header compression is a technique

used on SLIP and PPP links to reduce the size of the IP and TCP headers from the normal 40 bytes to around 3-6 bytes (typically). This is done by recognizing that most fields in these headers don't change from one segment to the next on a given connection, and the fields that do change often change by a small amount. This allows a flag byte to be sent indicating which fields have changed, and the changes are encoded as differences from the previous segment.

## Exercises

**29.1** A client connects to a server and no segments are lost. Which process, the client or server, completes its open of the connection first?

**29.2** A Net/3 system receives a SYN for a listening socket and the SYN segment also contains 50 bytes of data. What happens?

Continue the previous exercise assuming that the client does not retransmit the 50 bytes of data; **29.3** instead the client responds with a segment that acknowledges the server's SYN/ACK and contains a FIN. What happens?

A Net/3 client performs a passive open to a listening server. The server's response to the client's SYN is a segment with the expected SYN/ACK, but the segment also contains 50 bytes of data and the FIN flag. List the processing steps for the client's TCP. **29.4**

Figure 18.19 in Volume 1 and Figure 14 in RFC 793 both show four segments exchanged during a simultaneous close. But if we trace a simultaneous close between two Net/3 systems, or if we watch the close sequence following a self-connect on a Net/3 system, we see six segments, not four. The extra two **29.5**

segments are a retransmission of the FIN by each end when the other's FIN is received. Where is the bug and what is the fix?

Page 72 of RFC 793 says that when data in the send buffer is acknowledged by the other end "Users should receive positive acknowledgments for buffers which have been sent and fully acknowledged (i.e., send buffer should be returned with 'ok' response)." Does Net/3 provide this notification?

What effect do the options defined in  
**29.7** RFC 1323 have on TCP header compression?

What effect does the Net/3 assignment of the IP identification field have on TCP header compression?  
**29.8**

---

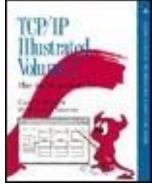
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 30. TCP User Requests

[Section 30.1. Introduction](#)

[Section 30.2. tcp\\_usrreq Function](#)

[Section 30.3. tcp\\_attach Function](#)

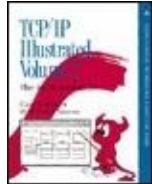
[Section 30.4. tcp\\_disconnect Function](#)

[Section 30.5. tcp\\_usrclosed Function](#)

[Section 30.6. tcp\\_ctloutput Function](#)

[Section 30.7. Summary](#)



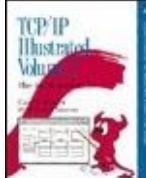


[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

### 30.1 Introduction

This chapter looks at the TCP user-request function `tcp_usrreq`, which is called as the protocol's `pr_usrreq` function to handle many of the system calls that reference a TCP socket. We also look at `tcp_ctloutput`, which is called when the process calls `setsockopt` for a TCP socket.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

## 30.2 tcp\_usrreq Function

TCP's user-request function is called for a variety of operations. [Figure 30.1](#) shows the beginning and end of `tcp_usrreq`. The body of the switch is shown in following figures. The function arguments, some of which differ depending on the request, are described in [Figure 15.17](#).

**Figure 30.1. Body of `tcp_usrreq` function.**

---

```

45 int
46 tcp_usrreq(so, req, m, nam, control)
47 struct socket *so;
48 int , req;
49 struct mbuf *m, *nam, *control;
50 {
51     struct inpcb *inp;
52     struct tcpcb *tp;
53     int s;
54     int error = 0;
55     int ostate;
56     if (req == PRU_CONTROL)
57         return (in_control(so, (int) m, (caddr_t) nam,
58                             (struct ifnet *) control));
59     if (control && control->m_len) {
60         m_freem(control);
61         if (m)
62             m_freem(m);
63         return (EINVAL);
64     }
65     s = splnet();
66     inp = sotoinpcb(so);
67     /*
68      * When a TCP is attached to a socket, then there will be
69      * a (struct inpcb) pointed at by the socket, and this
70      * structure will point at a subsidiary (struct tcpcb).
71      */
72     if (inp == 0 && req != PRU_ATTACH) {
73         splx(s);
74         return (EINVAL); /* XXX */
75     }
76     if (inp) {
77         tp = intotcpb(inp);
78         /* WHAT IF TP IS 0? */
79         ostate = tp->t_state;
80     } else
81         ostate = 0;
82     switch (req) {
83         /* switch cases */
84     default:
85         panic("tcp_usrreq");
86     }
87     if (tp && (so->so_options & SO_DEBUG))
88         tcp_trace(TA_USER, ostate, tp, (struct tciphdr *) 0, req);
89     splx(s);
90     return (error);
91 }

```

---

tcp\_usrreq.c

## in\_control processes ioctl requests

45-58

The PRU\_CONTROL request is from the ioctl sys call. The function in\_control processes the request.

completely.

## Control information is invalid

59-64

A call to sendmsg specifying control information invalid for a TCP socket. If this happens, the m<sub>i</sub> are released and EINVAL is returned.

65-66

This remainder of the function executes at spln. This is overly conservative locking to avoid sprinkling the individual case statements with c to splnet when the calls are really necessary. A mentioned with [Figure 23.15](#), setting the procedure priority to splnet only stops a software interrupt from causing the IP input routine to be executed (which could call tcp\_input). It does not prevent the interface layer from accepting incoming packets and placing them onto IP's input queue.

The pointer to the Internet PCB is obtained from the socket structure pointer. The only time the resulting PCB pointer is allowed to be a null pointer is when the PRU\_ATTACH request is issued, which occurs in response to the socket system call.

If `inp` is nonnull, the current connection state is saved in `ostate` for the call to `tcp_trace` at the end of the function.

We now discuss the individual case statements. `PRU_ATTACH` request, shown in [Figure 30.2](#), is issued by the socket system call and by `sonewc` when a connection request arrives for a listening socket ([Figure 28.7](#)).

**Figure 30.2. `tcp_usrreq` function: `PRU_ATTACH` and `PRU_DETACH` requests.**

```

83     /*
84      * TCP attaches to socket via PRU_ATTACH, reserving space,
85      * and an internet control block.
86      */
87     case PRU_ATTACH:
88         if (inp) {
89             error = EISCONN;
90             break;
91         }
92         error = tcp_attach(so);
93         if (error)
94             break;
95         if ((so->so_options & SO_LINGER) && so->so_linger == 0)
96             so->so_linger = TCP_LINGERTIME;
97         tp = sototcpb(so);
98         break;

99     /*
100      * PRU_DETACH detaches the TCP protocol from the socket.
101      * If the protocol state is non-embryonic, then can't
102      * do this directly: have to initiate a PRU_DISCONNECT,
103      * which may finish later; embryonic TCB's can just
104      * be discarded here.
105      */
106     case PRU_DETACH:
107         if (tp->t_state > TCPS_LISTEN)
108             tp = tcp_disconnect(tp);
109         else
110             tp = tcp_close(tp);
111         break;

```

tcp\_usrreq.c

## PRU\_ATTACH request

83-94

If the socket structure already points to a PCB, EISCONN is returned. `tcp_attach` completes the processing: it allocates and initializes the Internet PCB and the TCP control block.

95-96

If the `SO_LINGER` socket option is set, and the linger time is 0, it is set to 120 (`TCP_LINGERTIME`)

How can a socket option be set before the PRU\_ATTACH request is issued? It is impossible to set a socket option before calling socket, because sonewconn also issues the PRU\_ATTACH request. The PRU\_ATTACH request is issued after sonewconn copies the so\_options from the listening socket to the newly created socket. This code prevents a newly accepted connection from inheriting a linger time of 0 from the listening socket.

There is a bug here. The constant TCP\_LINGERTIME is initialized to 120 in the header tcp\_timer.h with the comment "linger most 2 minutes." But the so\_linger value becomes the final argument to the kernel's tsleep function (called from soclose), which becomes the final argument to the kernel's timeout function and is in clock ticks, not seconds. If the system's clock-tick frequency (hz) is 100, this value for the linger time is 1.2 seconds, not 2 minutes.

97

tp is now set to the pointer to the socket's TCP control block. This is required at the end, in case the SO\_DEBUG socket option is set.

## **PRU\_DETACH request**

99-111

The close system call issues the PRU\_DETACH request if the PRU\_DISCONNECT request fails. If the connection has not been completed (the connection state is less than ESTABLISHED), nothing needs to be sent to the other end. But if the connection has been established, `tcp_disconnect` initiates TCP's connection-close sequence (e.g., any pending data is sent, followed by a FIN).

The test for the state being greater than LISTEN is incorrect, because if the state is SYN\_SENT or SYN\_RCVD, both of which are greater than LISTEN, `tcp_disconnect` just calls `tcp_close`. This case could be simplified by just calling `tcp_disconnect`.

[Figure 30.3](#) shows the processing for the bind and listen system calls.

**Figure 30.3. `tcp_usrreq` function: PRU\_BIND and PRU\_LISTEN requests.**

---

```

112      /*
113       * Give the socket an address.
114       */
115     case PRU_BIND:
116         error = in_pcbbind(inp, nam);
117         if (error)
118             break;
119         break;

120      /*
121       * Prepare to accept connections.
122       */
123     case PRU_LISTEN:
124         if (inp->inp_lport == 0)
125             error = in_pcbbind(inp, (struct mbuf *) 0);
126         if (error == 0)
127             tp->t_state = TCPS_LISTEN;
128         break;

```

---

tcp\_usrreq.c

## 112-119

All the work for a PRU\_BIND request is done by in\_pcbbind.

## 120-128

For the PRU\_LISTEN request, if the socket has been bound with a local port, in\_pcbbind assigns one automatically. This is rare, since most servers explicitly bind their well-known port, although I (remote procedure call) servers typically bind a ephemeral port and then register the port with *Port Mapper*. (Section 29.4 of Volume 1 describes the Port Mapper.) The connection state is set to LISTEN. This is the main purpose of listen: to set the socket's state so that incoming connections are accepted (i.e., a passive open).

[Figure 30.4](#) shows the processing for the conn

system call: an active open normally initiated by client.

## Figure 30.4. `tcp_usrreq` function: PRU\_CONN request.

```
129      /*          _____tcp_usrreq.c
130      * Initiate connection to peer.
131      * Create a template for use in transmissions on this connection.
132      * Enter SYN_SENT state, and mark socket as connecting.
133      * Start keepalive timer, and seed output sequence space.
134      * Send initial segment on connection.
135      */
136 case PRU_CONNECT:
137     if (inp->inp_lport == 0) {
138         error = in_pcbbind(inp, (struct mbuf *) 0);
139         if (error)
140             break;
141     }
142     error = in_pcbservice(inp, nam);
143     if (error)
144         break;
145
146     tp->t_template = tcp_template(tp);
147     if (tp->t_template == 0) {
148         in_pcbservice_disconnect(inp);
149         error = ENOBUFS;
150         break;
151     }
152     /* Compute window scaling to request. */
153     while (tp->request_r_scale < TCP_MAX_WINSHIFT &&
154            (TCP_MAXWIN << tp->request_r_scale) < so->so_rcv.sb_hiwat)
155         tp->request_r_scale++;
156     soisconnecting(so);
157     tcpstat.tcp_s_connattempt++;
158     tp->t_state = TCPS_SYN_SENT;
159     tp->t_timer[TCPTV_KEEP] = TCPTV_KEEP_INIT;
160
161     tp->iss = tcp_iss;
162     tcp_iss += TCP_ISSINCR / 2;
163     tcp_sendseqinit(tp);
164
165     error = tcp_output(tp);
166     break;
```

## Assign ephemeral port

129-141

If the socket has not been bound with a local port, in\_pcbbind assigns one automatically. This is typical for clients, which normally don't care about the value of the local port.

## Connect PCB

142-144

in\_pcbbind acquires a route to the destination, determines the outgoing interface, and verifies the socket pair is unique.

## Initialize IP and TCP headers

145-150

tcp\_template allocates an mbuf for a copy of the IP and TCP headers, and it initializes both headers with as much information as possible. The only reason for this function to fail is for the kernel to run out of mbufs.

## Calculate window scale factor

151-154

The window scale value for the receive buffer is calculated: 65535 (TCP\_MAXWIN) is left shifted until the value is greater than or equal to the size of the receive buffer (so\_rcv.sb\_hiwat). The resulting shift count (between 0 and 14) is the scale factor that will be sent in the SYN. (We saw identical code in [Figure 28.7](#) that was executed during a passive open.) Since the window scale option is sent in the SYN resulting from a connect, the process must set the SO\_RCVBUF socket option before calling connect, or the default buffer size will be used (tcp\_recvspace from [Figure 24.3](#)).

## Set socket and connection state

155-158

soisconnecting sets the appropriate bits in the socket's state variable, and the state of the TCI connection is set to SYN\_SENT. This causes the code to call tcp\_output that follows to send the SYN (see the tcp\_outflags value in [Figure 24.16](#)). The connection-establishment timer is initialized to 3 seconds. tcp\_output will also set the retransmission timer for the SYN, as shown in [Figure 25.15](#).

## Initialize sequence numbers

159-161

The initial send sequence number is copied from the global `tcp_iss`. This global is then incremented by 64,000 (`TCP_ISSINCR` divided by 2). We saw this same handling of `tcp_iss` when the ISS was initialized after a listening server received a SYN ([Figure 28.17](#)). The send sequence numbers are then initialized by `tcp_sendseqinit`.

## Send initial SYN

162

`tcp_output` sends the initial SYN to initiate the connection. A local error (for example, out of mbufs or no route to destination) is returned by `tcp_output`, which becomes the return value from `tcp_usrreq`, which is returned to the process.

[Figure 30.5](#) shows the processing for the `PRU_CONNECT2`, `PRU_DISCONNECT`, and `PRU_ACCEPT` requests.

**Figure 30.5. `tcp_usrreq` function:  
`PRU_CONNECT2`, `PRU_DISCONNECT`, and  
`PRU_ACCEPT` requests.**

```

164      /*
165       * Create a TCP connection between two sockets.
166       */
167     case PRU_CONNECT2:
168         error = EOPNOTSUPP;
169         break;
170
171     /*
172      * Initiate disconnect from peer.
173      * If connection never passed embryonic stage, just drop;
174      * else if don't need to let data drain, then can just drop anyway,
175      * else have to begin TCP shutdown process: mark socket disconnecting,
176      * drain unread data, state switch to reflect user close, and
177      * send segment (e.g. FIN) to peer. Socket will be really disconnected
178      * when peer sends FIN and acks ours.
179
180      * SHOULD IMPLEMENT LATER PRU_CONNECT VIA REALLOC TCPCB.
181
182     case PRU_DISCONNECT:
183         tp = tcp_disconnect(tp);
184         break;
185
186     /*
187      * Accept a connection. Essentially all the work is
188      * done at higher levels; just return the address
189      * of the peer, storing through addr.
190
191     case PRU_ACCEPT:
192         in_setpeeraddr(inp, nam);
193         break;

```

-tcp\_usrreq.c

## 164-169

The PRU\_CONNECT2 request, a result of the socketpair system call, is invalid for the TCP protocol.

## 170-183

The close system call issues the PRU\_DISCONNECT request. If the connection has been established FIN must be sent and the normal TCP close sequence followed. This is done by tcp\_disconnect.

The comment beginning with "SHOULD IMPLEMENT" refers to the fact that a socket that encounters an error cannot be reused. For

example, if a client issues a connect and receives an error, it cannot issue another connect on the same socket. Instead, the socket with the error must be closed, a new socket created with socket, and the connect issued on the new socket.

## 184-191

All the work associated with the accept system call is done by the socket layer and the protocol layer. The PRU\_ACCEPT request just returns the IP address and port number of the peer to the process.

The PRU\_SHUTDOWN, PRU\_RCVD, and PRU\_SE requests are processed in [Figure 30.6](#).

**Figure 30.6. `tcp_usrreq` function:  
PRU\_SHUTDOWN, PRU\_RCVD, and PRU\_SE  
requests.**

```

192      /*
193       * Mark the connection as being incapable of further output.
194       */
195     case PRU_SHUTDOWN:
196         socantsendmore(so);
197         tp = tcp_usrclosed(tp);
198         if (tp)
199             error = tcp_output(tp);
200         break;
201
202     /*
203      * After a receive, possibly send window update to peer.
204      */
205     case PRU_RCVD:
206         (void) tcp_output(tp);
207         break;
208
209     /*
210      * Do a send by putting data in output queue and updating urgent
211      * marker if URG set. Possibly send more data.
212      */
213     case PRU_SEND:
214         sbappend(&so->so_snd, m);
215         error = tcp_output(tp);
216         break;

```

tcp\_usrreq.c

## PRU\_SHUTDOWN request

192-200

This request is issued by soshutdown when the process calls shutdown to prevent any further output. socantsendmore sets the socket's flags prevent any future output. tcp\_usrclosed sets the connection state according to [Figure 24.15](#). tcp\_output attempts to send the FIN, but if there is still pending data to send to the other end, that data is sent before the FIN is sent.

## PRU\_RCVD request

201-206

This request is issued by soreceive after the process has read data from the socket's receive buffer. TCP needs to know about this since the receive buffer may now have enough room to cause the advertised window to increase. tcp\_output determine whether a window update segment should be sent.

## **PRU\_SEND request**

207-214

In [Figure 23.14](#) we showed how the five write functions ended up issuing this request. sbappend adds the data to the socket's send buffer (which must wait until acknowledged by the other end) and tcp\_output sends a segment, if possible.

[Figure 30.7](#) shows the processing of the PRU\_ABORT and PRU\_SENSE requests.

**Figure 30.7. `tcp_usrreq` function: PRU\_ABORT and PRU\_SENSE requests.**

```
215     /*  
216      * Abort the TCP.  
217      */  
218     case PRU_ABORT:  
219         tp = tcp_drop(tp, ECONNABORTED);  
220         break;  
221     case PRU_SENSE:  
222         ((struct stat *) m)->st_blksize = so->so_snd.sb_hiwat;  
223         (void) splx(s);  
224         return (0);
```

*tcp\_usrreq.c*

## PRU\_ABORT request

215-220

A PRU\_ABORT request is issued for a TCP socket's soclose if the socket is a listening socket (e.g., server) and if there are pending connections for the server that have already initiated or completed a three-way handshake, but have not been accepted by the server yet. `tcp_drop` sends an RST if the connection is synchronized.

## PRU\_SENSE request

221-224

The fstat system call generates the PRU\_SENSE request. TCP returns the size of the send buffer in the `st_blksize` element of the stat structure.

Figure 30.8 shows the PRU\_RCVOOB request,

issued by soreceive when the process issues a system call specifying the MSG\_OOB flag to read out-of-band data.

## Figure 30.8. `tcp_usrreq` function: PRU\_RCV request.

```
225     case PRU_RCVOOB:                                tcp_usrreq.c
226         if ((so->so_oobmark == 0 &&
227             (so->so_state & SS_RCVATMARK) == 0) ||
228             so->so_options & SO_OOBINLINE ||
229             tp->t_oobflags & TCPOOB_HADDATA) {
230             error = EINVAL;
231             break;
232         }
233         if ((tp->t_oobflags & TCPOOB_HAVEDATA) == 0) {
234             error = EWOULDBLOCK;
235             break;
236         }
237         m->m_len = 1;
238         *mtod(m, caddr_t) = tp->t_iobc;
239         if (((int) nam & MSG_PEEK) == 0)
240             tp->t_oobflags ^= (TCPOOB_HAVEDATA | TCPOOB_HADDATA);
241         break;

```

*tcp\_usrreq.c*

## Verify that reading out-of-band data is appropriate

225-232

It is an error for the process to try to read out-of-band data if any one of the following three conditions is true:

- 1. if the socket's out-of-band mark is 0**

**(so\_oobmark) and the socket is not at t  
mark (the SS\_RCVATMARK flag is not se  
or**

- if the SO\_OOBINLINE socket option is set, or
- if the TCPOOB\_HADDATA flag is set for the connection (i.e., the connection did have an out-of-band byte, but it has already been read).

The error EINVAL is returned if any one of these conditions is true.

## Check that out-of-band byte has arrived

233-236

If none of the three conditions above is true, but the TCPOOB\_HAVEDATA flag is false, this indicates that TCP has received an urgent mode notification from the other end, but the byte whose sequence number is 1 less than the urgent pointer has not yet been received yet ([Figure 29.17](#)). The error EWOULDBLOCK is returned. It is possible for TCP to send an urgent notification with an urgent offset referencing a byte that the sender has not been able to send yet. Figure 26.7 of Volume 1 shows an example of this scenario, which often happens.

the sender's data transmission has been stopped by a zero-window advertisement.

## Return out-of-band byte

237-238

The single byte of out-of-band data that was stored in t\_iobc by tcp\_pullofband is returned to the process.

## Flip flags

239-241

If the process is actually reading the out-of-band byte (as compared to peeking at it with the MSG\_PEEK flag), this exclusive OR turns the HAVE flag off and the HAD flag on. We are guaranteed this point in the case statement that the HAVE flag is set and the HAD flag is cleared. The purpose of the HAD flag is to prevent the process from trying to read the out-of-band byte more than once. Once the HAD flag is set, it is not cleared until a new urgent pointer is received from the other end (Figure 29.17).

The reason for this hard-to-understand exclus  
OR, instead of the simpler

tp->t\_oobflags = TCPOOB\_HADDA]

is to allow additional bits in t\_oobflags to be used. Net/3, however, only uses the 2 bits that we've described.

The PRU\_SENDOOB request, shown in [Figure 3](#) is issued by sosend when the process writes data and specifies the MSG\_OOB flag.

### Figure 30.9. `tcp_usrreq` function: PRU\_SENDOOB request.

```
242     case PRU_SENDOOB:
243         if (sbspace(&so->so_snd) < -512) {
244             m_freem(m);
245             error = ENOBUFS;
246             break;
247         }
248         /*
249          * According to RFC961 (Assigned Protocols),
250          * the urgent pointer points to the last octet
251          * of urgent data. We continue, however,
252          * to consider it to indicate the first octet
253          * of data past the urgent section.
254          * Otherwise, snd_up should be one lower.
255          */
256         sbappend(&so->so_snd, m);
257         tp->snd_up = tp->snd_una + so->so_snd.sb_cc;
258
259         tp->t_force = 1;
260         error = tcp_output(tp);
261         tp->t_force = 0;
262         break;
```

## Check for room and append to send buffer

242-247

The process is allowed to exceed the size of the send buffer by up to 512 bytes when sending out-of-band data. The socket layer is more permissive than the TCP layer, allowing out-of-band data to exceed the size of the send buffer by 1024 bytes ([Figure 16.24](#)). The sbappend function adds the data to the end of the send buffer.

## Calculate urgent pointer

248-257

The urgent pointer (snd\_up) points to the byte following the final byte from the write request. The code showed this in [Figure 26.30](#), assuming the process writes 3 bytes of data with the MSG\_OOB flag set and that the send buffer was empty. Realize that if the process writes more than 1 byte of data with the MSG\_OOB flag set, only the final byte is considered the out-of-band byte when the data is received by a Berkeley-derived system.

## Force TCP output

258-261

t\_force is set to 1 and tcp\_output is called. This causes a segment to be sent with the URG flag and with a nonzero urgent offset, even if no data can be sent because of a zero-window advertisement. Figure 26.7 of Volume 1 shows transmission of an urgent segment into a closed window.

The final three requests are shown in [Figure 30](#)

**Figure 30.10. `tcp_usrreq` function:  
`PRU_SOCKADDR`, `PRU_PEERADDR`, and  
`PRU_SLOWTIMO` requests.**

```
262     case PRU_SOCKADDR:
263         in_setsockaddr(inp, nam);
264         break;
265     case PRU_PEERADDR:
266         in_setpeeraddr(inp, nam);
267         break;
268     /*
269      * TCP slow timer went off; going through this
270      * routine for tracing's sake.
271      */
272     case PRU_SLOWTIMO:
273         tp = tcp_timers(tp, (int)nam);
274         req |= (int)nam << 8; /* for debug's sake */
275         break;
```

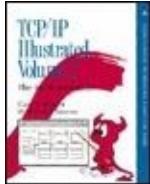
262-267

The `getsockname` and `getpeername` system cal

issue the PRU\_SOCKADDR and PRU\_PEERADDR requests, respectively. The functions in\_setsockaddr and in\_setpeeraddr fetch the information from the PCB, storing the result in addr argument.

268-275

The PRU\_SLOWTIMO request is issued by the tcp\_slowtimo function. As the comment indicates, the only reason tcp\_slowtimo doesn't call tcp\_timers directly is to allow the timer expiration to be traced by the call to tcp\_trace at the end of the function ([Figure 30.1](#)). For the trace record to show which one of the four TCP timer counters expired, tcp\_slowtimo passes the index into the t\_timer array ([Figure 25.1](#)) as the nam argument and this is left shifted 8 bits and logically ORed with the request value (req). The trpt program knows about this hack and handles it accordingly.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

### 30.3 tcp\_attach Function

The `tcp_attach` function is called by `tcp_usrreq` to process the `PRU_ATTACH` request (i.e., when the socket system call is issued or when a new connection request arrives for a listening socket). Figure 30.11 shows the code.

**Figure 30.11. `tcp_attach` function: create a new TCP socket.**

---

```

361 int
362 tcp_attach(so)
363 struct socket *so;
364 {
365     .
366     struct tcpcb *tp;
367     struct inpcb *inp;
368     int     error;
369
370     if (so->so_snd.sb_hiwat == 0 || so->so_rcv.sb_hiwat == 0) {
371         error = soreserve(so, tcp_sendspace, tcp_recvspace);
372         if (error)
373             return (error);
374     }
375     error = in_pcalloc(so, &tcb);
376     if (error)
377         return (error);
378     inp = sotoinpcb(so);
379     tp = tcp_newtcpcb(inp);
380     if (tp == 0) {
381         int     nofd = so->so_state & SS_NOFDREF; /* XXX */
382         so->so_state &= ~SS_NOFDREF; /* don't free the socket yet */
383         in_pcbdetach(inp);
384         so->so_state |= nofd;
385         return (ENOBUFS);
386     }
387     tp->t_state = TCPS_CLOSED;
388     return (0);
389 }
```

---

tcp\_usrreq.c

## Allocate space for send buffer and receive buffer

361-372

If space has not been allocated for the socket's send and receive buffers, sbreserve sets them both to 8192, the default values of the global variables `tcp_sendspace` and `tcp_recvspace` ([Figure 24.3](#)).

Whether these defaults are adequate depends on the MSS for each direction

of the connection, which depends on the MTU. For example, [Comer and Lin 1994] show that anomalous behavior occurs if the send buffer is less than three times the MSS, which drastically reduces performance. Some implementations have much higher defaults, such as 61,444 bytes, realizing the effect these defaults have on performance, especially with higher MTUs (e.g., FDDI and ATM).

## Allocate Internet PCB and TCP control block

373-377

`in_pcalloc` allocates an Internet PCB and `tcp_newtcpcb` allocates a TCP control block and links it to the PCB.

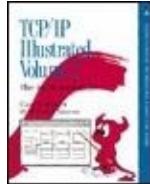
378-384

The code with the comment XXX is executed if the call to malloc in `tcp_newtcpcb` fails. Remember that the PRU\_ATTACH request is issued as a result

of the socket system call, and when a connection request arrives for a listening socket (sonewconn). In the latter case the socket flag SS\_NOFDREF is set. If this flag is left on, the call to sofref by in\_pcboffload releases the socket structure. As we saw in tcp\_input, this structure should not be released until that function is done with the received segment (the dropsocket flag in [Figure 29.27](#)). Therefore the current value of the SS\_NOFDREF flag is saved in the variable nofd when in\_pcboffload is called, and reset before tcp\_attach returns.

385-386

The TCP connection state is initialized to CLOSED.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

### 30.4 `tcp_disconnect` Function

`tcp_disconnect`, shown in Figure 30.12, initiates a TCP disconnect.

**Figure 30.12. `tcp_disconnect` function:  
initiate TCP disconnect.**

```
396 struct tcpcb *          ——————tcp_usrreq.c
397 tcp_disconnect(tp)
398 struct tcpcb *tp;
399 {
400     struct socket *so = tp->t_inpcb->inp_socket;
401     if (tp->t_state < TCPS_ESTABLISHED)
402         tp = tcp_close(tp);
403     else if ((so->so_options & SO_LINGER) && so->so_linger == 0)
404         tp = tcp_drop(tp, 0);
405     else {
406         soisdisconnecting(so);
407         sbflush(&so->so_rcv);
408         tp = tcp_usrclosed(tp);
409         if (tp)
410             (void) tcp_output(tp);
411     }
412     return (tp);
413 }
```

## Connection not yet synchronized

396-402

If the socket is not yet in the ESTABLISHED state (i.e., LISTEN, SYN\_SENT, or SYN\_RCVD), `tcp_close` just releases the PCB and the TCP control block. Nothing needs to be sent to the other end since the connection has not been synchronized.

## Hard disconnect

403-404

If the connection is synchronized, the `SO_LINGER` socket option is set, and the linger time (`so_linger`) is set to 0, the connection is dropped by `tcp_drop`. This sets the connection state to CLOSED, sends an RST to the other end, and releases the PCB and TCP control block. The connection does not pass through the TIME\_WAIT state. The call to close that caused the PRU\_DISCONNECT request will discard any data still in the send or receive

buffers.

If the SO\_LINGER socket option has been set with a nonzero linger time, it is handled by soclose.

## Graceful disconnect

405-406

This code is executed when the connection has been synchronized but the SO\_LINGER option either was not set or was set with a nonzero linger time. TCP's normal connection termination steps must be followed. soisdisconnecting sets the socket's state.

## Discard pending receive data

407

Any pending data in the receive buffer is discarded by sbflush, since the process has closed the socket. The send buffer is left alone, however, and tcp\_output will try to send what remains. We say "try" because

there's no guarantee that the data still to be sent will be transmitted successfully. The other end might crash before it receives and acknowledges the data, or even if the TCP module at the other end receives and acknowledges the data, the system might crash before the application at the other end reads the data. Since the local process has closed the socket, if TCP gives up trying to send what remains in the send buffer (because its retransmission timer finally expires), there is no way to notify the process of the error.

## Change connection state

408-410

tcp\_usrclosed moves the connection into the next state, based on the current state. This normally moves the connection to the FIN\_WAIT\_1 state, since the connection is typically closed from the ESTABLISHED state. We'll see that tcp\_usrclosed always returns the current control block pointer (tp), since the state must be synchronized to get to this point in the code, so

`tcp_output` is always called to send a segment. If the connection moves from the `ESTABLISHED` to the `FIN_WAIT_1` state, this causes a FIN to be sent.

---

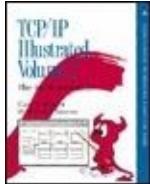
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

### 30.5 `tcp_usrclosed` Function

This function, shown in [Figure 30.13](#), is called from `tcp_disconnect` and when the `PRU_SHUTDOWN` request is processed.

**Figure 30.13. `tcp_usrclosed` function:  
move connection to next state, based on  
process close.**

---

```

424 struct tcpcb *
425 tcp_usrclosed(tp)
426 struct tcpcb *tp;
427 {
428     switch (tp->t_state) {
429         case TCPS_CLOSED:
430         case TCPS_LISTEN:
431         case TCPS_SYN_SENT:
432             tp->t_state = TCPS_CLOSED;
433             tp = tcp_close(tp);
434             break;
435
436         case TCPS_SYN_RECEIVED:
437         case TCPS_ESTABLISHED:
438             tp->t_state = TCPS_FIN_WAIT_1;
439             break;
440
441         case TCPS_CLOSE_WAIT:
442             tp->t_state = TCPS_LAST_ACK;
443             break;
444     }
445     if (tp && tp->t_state >= TCPS_FIN_WAIT_2)
446         soisdisconnected(tp->t_inpcb->inp_socket);
447     return (tp);
448 }
```

---

tcp\_usrreq.c

## Simple close when SYN not received

429-434

If a SYN has not been received on the connection, a FIN need not be sent. The new state is CLOSED and `tcp_close` releases the Internet PCB and the TCP control block.

## Move to FIN\_WAIT\_1 state

435-438

In the SYN\_RCVD and ESTABLISHED

states, the new state is FIN\_WAIT\_1, which causes the next call to `tcp_output` to send a FIN (the `tcp_outflags` value in Figure 24.16).

## Move to LAST\_ACK state

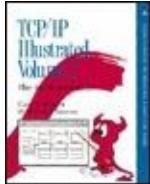
439-441

In the CLOSE\_WAIT state, the close moves the connection into the LAST\_ACK state. The next call to `tcp_output` will cause a FIN to be sent.

443-444

If the connection state is either FIN\_WAIT\_2 or TIME\_WAIT, `soisdisconnected` marks the socket state appropriately.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

### 30.6 tcp\_ctloutput Function

The `tcp_ctloutput` function is called by the `getsockopt` and `setsockopt` system calls when the descriptor argument refers to a TCP socket and when the level is not `SOL_SOCKET`. [Figure 30.14](#) shows the two socket options supported by TCP.

**Figure 30.14. Socket options supported by TCP.**

optname	Variable	Access	Description
TCP_NODELAY	t_flags	read, write	Nagle algorithm (Figure 26.8)
TCP_MAXSEG	t_maxseg	read, write	maximum segment size TCP will send

[Figure 30.15](#) shows the first part of the function.

## Figure 30.15. `tcp_ctloutput` function: first part.

```
284 int  
285 tcp_ctloutput(op, so, level, optname, mp)  
286 int     op;  
287 struct socket *so;  
288 int     level, optname;  
289 struct mbuf **mp;  
290 {  
291     int     error = 0, s;  
292     struct inpcb *inp;  
293     struct tcpcb *tp;  
294     struct mbuf *m;  
295     int     i;  
296     s = splnet();  
297     inp = sotoinpcb(so);  
298     if (inp == NULL) {  
299         splx(s);  
300         if (op == PRCO_SETOPT && *mp)  
301             (void) m_free(*mp);  
302         return (ECONNRESET);  
303     }  
304     if (level != IPPROTO_TCP) {  
305         error = ip_ctloutput(op, so, level, optname, mp);  
306         splx(s);  
307         return (error);  
308     }  
309     tp = intotcpcb(inp);  
-----tcp_usrreq.c-----tcp_usrreq.c
```

296-303

The processor priority is set to `splnet` while the function executes, and `inp` points to the Internet PCB for the socket. If `inp` is null, the mbuf is released if the operation was to set a socket option, and an error is returned.

304-308

If the `level` (the second argument to the

getsockopt and setsockopt system calls) is not IPPROTO\_TCP, the command is for some other protocol (i.e., IP). For example, it is possible to create a TCP socket and set the IP source routing socket option. In this example IP processes the socket option, not TCP. ip\_ctloutput handles the command.

309

The command is for TCP, so tp is set to the TCP control block.

The remainder of the function is a switch with two cases: one for PRCO\_SETOPT (shown in [Figure 30.16](#)) and one for PRCO\_GETOPT (shown in [Figure 30.17](#)).

**Figure 30.16. tcp\_ctloutput function: set a socket option.**

---

```

310     switch (op) {
311         case PRCO_SETOPT:
312             m = *mp;
313             switch (optname) {
314                 case TCP_NODELAY:
315                     if (m == NULL || m->m_len < sizeof(int))
316                         error = EINVAL;
317                     else if (*mtod(m, int *) != tp->t_flags & TF_NODELAY)
318                         tp->t_flags |= TF_NODELAY;
319                     else
320                         tp->t_flags &= ~TF_NODELAY;
321                     break;
322
323                 case TCP_MAXSEG:
324                     if (m && (i = *mtod(m, int *)) > 0 && i <= tp->t_maxseg)
325                         tp->t_maxseg = i;
326                     else
327                         error = EINVAL;
328                     break;
329
330                 default:
331                     error = ENOPROTOOPT;
332                     break;
333             }
334             if (m)
335                 (void) m_free(m);
336             break;

```

---

*tcp\_usrreq.c*

## Figure 30.17. `tcp_ctloutput` function: get a socket option.

---

```

335     case PRCO_GETOPT:
336         *mp = m = m_get(M_WAIT, MT_SOOPTS);
337         m->m_len = sizeof(int);
338
339         switch (optname) {
340             case TCP_NODELAY:
341                 *mtod(m, int *) = tp->t_flags & TF_NODELAY;
342                 break;
343             case TCP_MAXSEG:
344                 *mtod(m, int *) = tp->t_maxseg;
345                 break;
346             default:
347                 error = ENOPROTOOPT;
348                 break;
349             break;
350         }
351         splx(s);
352         return (error);
353     }

```

---

*tcp\_usrreq.c*

`m` is an mbuf containing the fourth argument to `setsockopt`. For both of the TCP options the mbuf must contain an integer value. If either the mbuf pointer is null, or the amount of data in the mbuf is less than the size of an integer, an error is returned.

## TCP\_NODELAY option

317-321

If the integer value is nonzero, the `TF_NODELAY` flag is set. This disables the Nagle algorithm in [Figure 26.8](#). If the integer value is 0, the Nagle algorithm is enabled (the default) and the `TF_NODELAY` flag is cleared.

## TCP\_MAXSEG option

322-327

A process can only decrease the MSS. When a TCP socket is created, `tcp_newtcpcb` initializes `t_maxseg` to its default of 512. When a SYN is received

from the other end with an MSS option, `tcp_input` calls `tcp_mss`, and `t_maxseg` can be set as high as the outgoing interface MTU (minus 40 bytes for the default IP and TCP headers), which is 1460 for an Ethernet. Therefore, after a call to `socket` but before a connection is established, a process can only decrease the MSS from its default of 512. After a connection is established, the process can decrease the MSS from whatever value was selected by `tcp_mss`.

4.4BSD was the first Berkeley release to allow the MSS to be set with a socket option. Prior releases only allowed a `getsockopt` for the MSS.

## Release mbuf

332-333

The mbuf chain is released.

[Figure 30.17](#) shows the processing for the `PRCO_GETOPT` command.

335-337

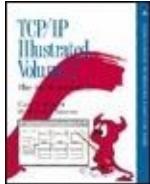
Both TCP socket options return an integer to the process, so m\_get obtains an mbuf and its length is set to the size of an integer.

339-341

TCP\_NODELAY returns the current status of the TF\_NODELAY flag: 0 if the flag is not set (the Nagle algorithm is enabled) or TF\_NODELAY if the flag is set.

342-344

The TCP\_MAXSEG option returns the current value of t\_maxseg. As we said in our discussion of the PRCO\_SETOPT command, the value returned depends whether the socket has been connected yet.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 30. TCP User Requests

### 30.7 Summary

The `tcp_usrreq` function is straightforward because most of the required processing is done by other functions. The `PRU_xxx` requests form the glue between the protocol-independent system calls and the TCP protocol processing.

The `tcp_ctloutput` function is also simple because only two socket options are supported by TCP: enable or disable the Nagle algorithm, and set or fetch the maximum segment size.

### Exercises

Now that we've covered all of TCP, list the processing steps and the TCP state transitions when a client goes through the normal steps of socket, connect, write (a request to the server), read (a reply from the server), and close. Do the same exercise for the server end.

If a process sets the SO\_LINGER socket option with a linger time of 0 and then calls close, we showed how `tcp_disconnect` is called, which causes an RST to be sent. What happens if a process sets this socket option with a linger time of 0 but is then killed by a signal instead of calling close? Is the RST segment still sent?

The description for TCP\_LINGERTIME in [Figure 25.4](#) is the "maximum #seconds for SO\_LINGER socket option." Given the code in [Figure 30.2](#), is this description correct?

A Net/3 client calls socket and connect to actively open a connection to a server. The server is reached through the client's default router. A total of 1,129 segments are

**30.4** sent by the client host to the server.

Assuming the route to the destination does not change, how many routing table lookups are done on the client host for this connection? Explain.

Obtain the sock program described in Appendix C of Volume 1. Run it as a sink server with a pause before reading (-P) and a large receive buffer. Then run the same program

**30.5** on another system as a source client. Watch the data with tcpdump. Verify that TCP's ACK-every-other-segment does not occur and that the only ACKs seen from the server are delayed ACKs.

Modify the SO\_KEEPALIVE socket option so that the parameters can be

**30.6** configured on a per-connection basis.

Read RFC 1122 to determine why it recommends that an implementation **30.7** should allow an RST to carry data. Modify the Net/3 code to implement this.

---

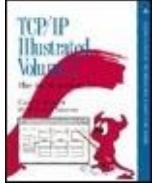
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 31. BPF: BSD Packet Filter

[Section 31.1. Introduction](#)

[Section 31.2. Code Introduction](#)

[Section 31.3. bpf\\_if Structure](#)

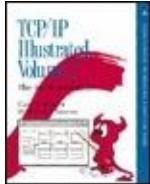
[Section 31.4. bpf\\_d Structure](#)

[Section 31.5. BPF Input](#)

[Section 31.6. BPF Output](#)

[Section 31.7. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 31. BPF: BSD Packet Filter

### 31.1 Introduction

The BSD Packet Filter (BPF) is a software device that "taps" network interfaces. A process accesses a BPF device by opening /dev/bpf0, /dev/bpf1, and so on. Each BPF device can be opened only by one process at a time.

Since each BPF device allocates 8192 bytes of buffer space, the system administrator typically limits the number of BPF devices. If open returns EBUSY, the device is in use, and a process tries the next device until the open succeeds.

The device is configured with several ioctl

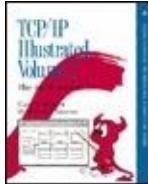
commands that associate the device with a network interface and install filters to receive incoming packets selectively. Packets are received by reading from the device, and packets are queued on the network interface by writing to the device.

We will use the term *packet* even though *frame* is more accurate, since BPF works at the data-link layer and includes the link-layer headers in the frames it sends and receives.

BPF works only with network interfaces that have been modified to support BPF. In [Chapter 3](#) we saw that the Ethernet, SLIP, and loopback drivers call `bpfattach`. This call configures the interface for access through the BPF devices. In this section we show how the BPF device driver is organized and how packets move between the driver and the network interfaces.

BPF is normally used as a diagnostic tool to examine the traffic on a locally attached network. The `tcpdump` program is the best example of such a tool and is described in Appendix A of Volume 1. Normally the user

is interested in packets between a given set of machines, or for a particular protocol, or even for a particular TCP connection. A BPF device can be configured with a filter that discards or accepts incoming packets according to a filter specification. Filters are specified as instructions to a pseudomachine. The details of BPF filters are not discussed in this text. For more information about filters, see `bpf(4)` and [[McCanne and Jacobson 1993](#)].



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 31. BPF: BSD Packet Filter

### 31.2 Code Introduction

The code for the portion of the BPF device driver that we describe resides in the two headers and one C file listed in [Figure 31.1](#).

**Figure 31.1. Files discussed in this chapter.**

File	Description
net/bpf.h	BPF constants
net/bpfdesc.h	BPF structures
net/bpf.c	BPF device support

### Global Variables

The global variables introduced in this chapter are shown in [Figure 31.2](#).

## Figure 31.2. Global variables introduced in this chapter.

Variable	Datatype	Description
bpf_iflist	struct bpf_if *	linked list of BPF-capable interfaces
bpf_dtab	struct bpf_d []	array of BPF descriptor structures
bpf_bufsize	int	default size of BPF buffers

## Statistics

[Figure 31.3](#) shows the two statistics collected in the bpf\_d structure for every active BPF device.

## Figure 31.3. Statistics collected in this chapter.

bpf_d member	Description
bd_rcount	#packets received from network interface
bd_dcount	#packets dropped because of insufficient buffer space

The remainder of this chapter is divided into four sections:

- BPF interface structures,
  - BPF device descriptors,
  - BPF input processing, and
  - BPF output processing.
- 

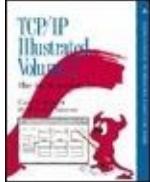
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



TCP/IP Illustrated, Volume 2: The Implementation  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 31. BPF: BSD Packet Filter

### 31.3 bpf\_if Structure

BPF keeps a list of the network interfaces that support BPF. Each interface is described by a `bpf_if` structure, and the global pointer `bpf_iflist` points to the first structure in the list. [Figure 31.4](#) shows a BPF interface structure.

**Figure 31.4. bpf\_if structure.**

```
67 struct bpf_if {  
68     struct bpf_if *bif_next;    /* list of all interfaces */  
69     struct bpf_d *bif_dlist;    /* descriptor list */  
70     struct bpf_if **bif_driverp; /* pointer into softc */  
71     u_int    bif_dlt;          /* link layer type */  
72     u_int    bif_hdrlen;       /* length of header (with padding) */  
73     struct ifnet *bif_ifp;      /* corresponding interface */  
74 };
```

`bif_next` points to the next BPF interface structure in the list. `bif_dlist` points to a list of BPF devices that have been opened and configured to tap this interface.

70

`bif_driverp` points to a `bpf_if` pointer stored in the `ifnet` structure of the tapped interface. When the interface is *not* tapped, `*bif_driverp` is null. When a BPF device is configured to tap an interface, `*bif_driverp` is changed to point back to the `bif_if` structure and tells the interface to begin passing packets to BPF.

71

The type of interface is saved in `bif_dlt`. The values for our example interfaces are shown in [Figure 31.5](#).

**Figure 31.5. `bif_dlt` values.**

<code>bif_dlt</code>	Description
<code>DLT_EN10MB</code>	10Mb Ethernet interface
<code>DLT_SLIP</code>	SLIP interface
<code>DLT_NULL</code>	loopback interface

72-74

Each packet accepted by BPF has a BPF header prepended to it. `bif_hdrlen` is the size of the header. Finally, `bif_ifp` points to the ifnet structure for the associated interface.

[Figure 31.6](#) shows the `bpf_hdr` structure that is prepended to every incoming packet.

### Figure 31.6. `bpf_hdr` structure.

```
122 struct bpf_hdr {  
123     struct timeval bh_tstamp;    /* time stamp */  
124     u_long    bh_caplen;        /* length of captured portion */  
125     u_long    bh_datalen;       /* original length of packet */  
126     u_short   bh_hdrlen;       /* length of bpf header (this struct plus  
127                                         alignment padding) */  
128 };
```

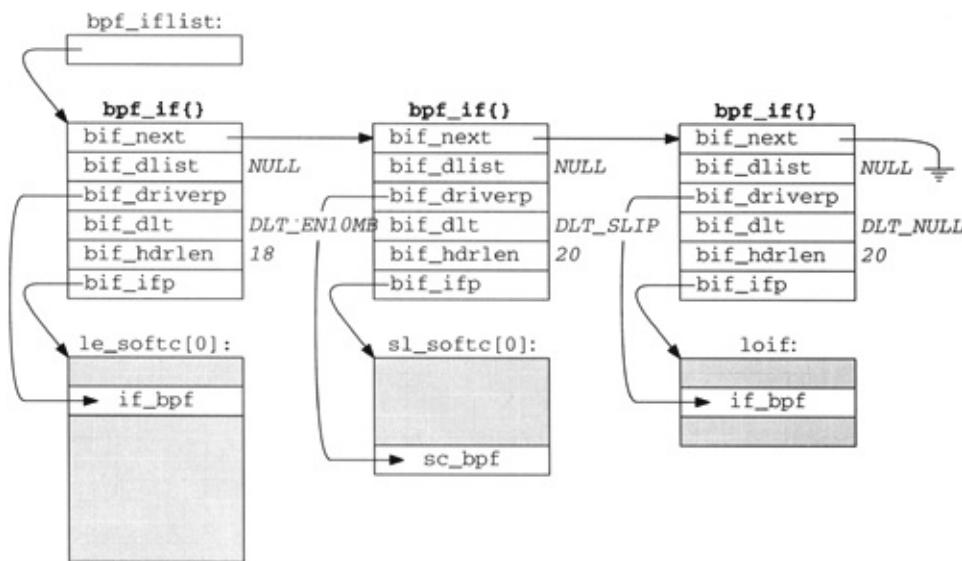
122-128

`bh_tstamp` records the time the packet was captured. `bh_caplen` is the number of bytes saved by BPF, and `bh_datalen` is the number of bytes in the original packet. `bh_headlen` is the size of the `bpf_hdr` structure plus any padding. This value should match `bif_hdrlen` for the receiving

interface and is used by processes to interpret the packets read from the BPF device.

[Figure 31.7](#) shows how `bpf_if` structures are connected to the `ifnet` structures for each of our three sample interfaces (`le_softc[0]`, `sl_softc[0]`, and `loif`).

**Figure 31.7. `bpf_if` and `ifnet` structures.**



Notice that `bif_driverp` points to the `if_bpf` and `sc_bpf` pointers in the network interfaces and *not* to the interface structures.

The SLIP device uses `sc_bpf`, instead of

the if\_bpf member. One reason might be that the SLIP BPF code was written before the if\_bpf member was added to the ifnet structure. The ifnet structure in Net/2 does not include a if\_bpf member.

The link-type and header-length members are initialized for all three interfaces according to the information passed by each driver in the call to bpfattach.

In [Chapter 3](#) we saw that bpfattach was called by the Ethernet, SLIP, and loop-back drivers. The linked list of BPF interface structures is built as each device driver calls bpfattach during initialization. The function is shown in [Figure 31.8](#).

## Figure 31.8. bpfattach function.

---

```

1053 void
1054 bpfattach(driverp, ifp, dlt, hdrlen)
1055 caddr_t *driverp;
1056 struct ifnet *ifp;
1057 u_int dlt, hdrlen;
1058 {
1059     struct bpf_if *bp;
1060     int i;
1061     bp = (struct bpf_if *) malloc(sizeof(*bp), M_DEVBUF, M_DONTWAIT);
1062     if (bp == 0)
1063         panic("bpfattach");
1064     bp->bif_dlist = 0;
1065     bp->bif_driverp = (struct bpf_if **) driverp;
1066     bp->bif_ifp = ifp;
1067     bp->bif_dlt = dlt;
1068     bp->bif_next = bpf_iflist;
1069     bpf_iflist = bp;
1070     *bp->bif_driverp = 0;
1071     /*
1072      * Compute the length of the bpf header. This is not necessarily
1073      * equal to SIZEOF_BPF_HDR because we want to insert spacing such
1074      * that the network layer header begins on a longword boundary (for
1075      * performance reasons and to alleviate alignment restrictions).
1076      */
1077     bp->bif_hdrlen = BPF_WORDALIGN(hdrlen + SIZEOF_BPF_HDR) - hdrlen;
1078     /*
1079      * Mark all the descriptors free if this hasn't been done.
1080      */
1081     if (!D_ISFREE(&bpfdtab[0]))
1082         for (i = 0; i < NBPFILTER; ++i)
1083             D_MARKFREE(&bpfdtab[i]);
1084     printf("bpf: %s%d attached\n", ifp->if_name, ifp->if_unit);
1085 }

```

---

bpf.c

## 1053-1063

bpfattach is called by each device driver that supports BPF. The first argument is the pointer saved in bif\_driverp (described with [Figure 31.4](#)). The second argument points to the ifnet structure of the interface. The third argument identifies the data-link type, and the fourth argument identifies the size of link-layer header passed with the packet. A new bpf\_if structure is allocated for the interface.

## **Initialize bpf\_if structure**

*1064-1070*

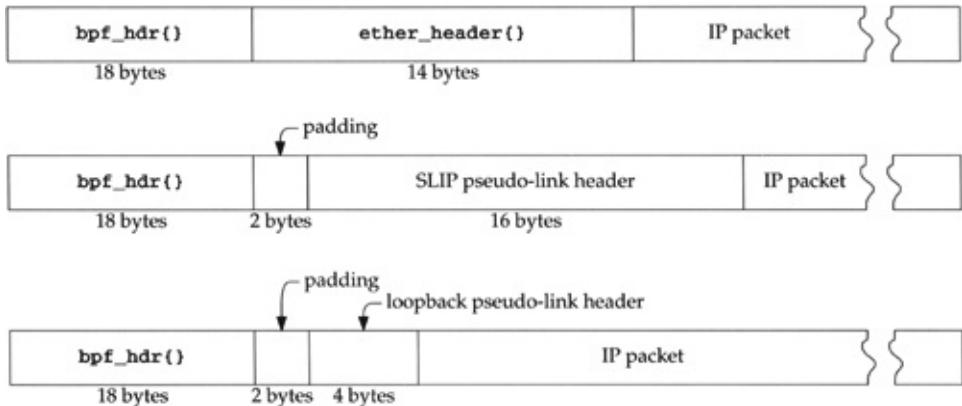
The bpf\_if structure is initialized from the arguments and inserted into the front of the BPF interface list, bpf\_iflist.

## **Compute BPF header size**

*1071-1077*

bif\_hdrlen is set to force the *network-layer* header (e.g., the IP header) to start on a longword boundary. This improves performance and avoids unnecessary alignment restrictions for the BPF filter. [Figure 31.9](#) shows the overall organization of the captured BPF packet for each of our three sample interfaces.

**Figure 31.9. BPF packet organization.**



The `ether_header` structure was described with [Figure 4.10](#), the SLIP pseudo-link header was described with [Figure 5.14](#), and the loopback pseudo-link header was described with [Figure 5.28](#).

Notice that the SLIP and loopback packets require 2 bytes of padding to force the IP header to appear on a 4-byte boundary.

## Initialize `bpf_dtab` table

*1078-1083*

This code initializes the BPF descriptor table, which is described with [Figure 31.10](#). The initialization occurs the first time `bpfattach` is called and is skipped thereafter.

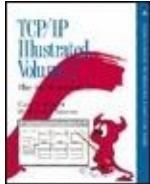
## Figure 31.10. bpf\_d structure.

```
45 struct bpf_d {  
46     struct bpf_d *bd_next;      /* Linked list of descriptors */  
47     caddr_t bd_sbuf;          /* store slot */  
48     caddr_t bd_hbuf;          /* hold slot */  
49     caddr_t bd_fbuf;          /* free slot */  
50     int     bd_slen;          /* current length of store buffer */  
51     int     bd_hlen;          /* current length of hold buffer */  
52     int     bd_bufsize;        /* absolute length of buffers */  
53     struct bpf_if *bd_bif;    /* interface descriptor */  
54     u_long   bd_rtout;        /* Read timeout in 'ticks' */  
55     struct bpf_insn *bd_filter; /* filter code */  
56     u_long   bd_rcount;        /* number of packets received */  
57     u_long   bd_dcount;        /* number of packets dropped */  
58     u_char   bd_promisc;       /* true if listening promiscuously */  
59     u_char   bd_state;         /* idle, waiting, or timed out */  
60     u_char   bd_immediate;    /* true to return on packet arrival */  
61     u_char   bd_pad;          /* explicit alignment */  
62     struct selinfo bd_sel;    /* bsd select info */  
63 };
```

## Print console message

1084-1085

A short message is printed to the console to announce that the interface has been configured for use by BPF.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 31. BPF: BSD Packet Filter

### 31.4 bpf\_d Structure

To begin tapping an interface, a process opens a BPF device and issues ioctl commands to select the interface, the read buffer size, and timeouts, and to specify a BPF filter. Each BPF device has an associated bpf\_d structure, shown in [Figure 31.10](#).

45-46

bpf\_d structures are placed on a linked list when more than one BPF device is attached to the same network interface. bd\_next points to the next structure in the list.

## Packet buffers

47-52

Each bpf\_d structure has two packet buffers associated with it. Incoming packets are always stored in the buffer attached to bd\_sbuf (the store buffer). The other buffer is either attached to bd\_fbuf (the free buffer), which means it is empty, or to bd\_hbuf (the hold buffer), which means it contains packets that are being read by a process. bd\_slen and bd\_hlen record the number of bytes saved in the store and hold buffer respectively.

When the store buffer becomes full, it is attached to bd\_hbuf and the free buffer is attached to bd\_sbuf. When the hold buffer is emptied, it is attached to bd\_fbuf. The macro ROTATE\_BUFFERS attaches the store buffer to bd\_hbuf, attaches the free buffer to bd\_sbuf, and clears bd\_fbuf. It is called when the store buffer becomes full, or when the process doesn't want to wait for more packets.

bd\_bufsize records the size of the two

buffers associated with the device. It defaults to 4096 (BPF\_BUFSIZE) bytes. The default value can be changed by patching the kernel, or bd\_bufsize can be changed for a particular BPF device with the BIOCSBLEN ioctl command. The BIOCGBLEN command returns the current value of bd\_bufsize, which can never exceed 32768 (BPF\_MAXBUFSIZE) bytes. There is also a minimum size of 32 (BPF\_MINBUFSIZE) bytes.

53-57

bd\_bif points to the bpf\_if structure associated with the BPF device. The BIOCSETIF command specifies the device. bd\_rtout is the number of clock ticks to delay while waiting for packets to appear. bd\_filter points to the BPF filter code for this device. Two statistics, which are available to a process through the BIOCSTATS command, are kept in bd\_rcount and bd\_dcount.

58-63

bd\_promisc is set with the BIOC PROMISC

command and causes the interface to operate in promiscuous mode. `bd_state` is unused. `bd_immediate` is set with the `BIOCIMMEDIATE` command and causes the driver to return each packet as it is received instead of waiting for the hold buffer to fill. `bd_pad` pads the `bpf_d` structure to a longword boundary, and `bd_sel` holds the `selinfo` structure for the select system call. We don't describe the use of select with a BPF device, but select itself is described in [Section 16.13](#).

## bpfopen Function

When `open` is called for a BPF device, the call is routed to `bpfopen` ([Figure 31.11](#)) for processing.

**Figure 31.11. `bpfopen` function.**

```
256 int  
257 bpfopen(dev, flag)  
258 dev_t dev;  
259 int flag;  
260 {  
261     struct bpf_d *d;  
262     if (minor(dev) >= NBPFILTER)  
263         return (ENXIO);  
264     /*  
265      * Each minor can be opened by only one process. If the requested  
266      * minor is in use, return EBUSY.  
267      */  
268     d = &bpf_dtab[minor(dev)];  
269     if (!D_ISFREE(d))  
270         return (EBUSY);  
271     /* Mark "free" and do most initialization. */  
272     bzero((char *) d, sizeof(*d));  
273     d->bd_bufsize = bpf_bufsize;  
274     return (0);  
275 }
```

bpf.c

## 256-263

The number of BPF devices is limited at compile time to NBPFILTER. The minor device number specifies the device and ENXIO is returned if it is too large. This happens when the system administrator creates more /dev/bpfx entries than the value NBPFILTER.

## Allocate bpf\_d structure

## 264-275

Only one process is allowed access to a BPF device at a time. If the bpf\_d structure is already active, EBUSY is

returned. Programs such as tcpdump try the next device when this error is returned. If the device is available, the entry in the bpf\_dtab table specified by the minor device number is cleared and the size of the packet buffers is set to the default value.

## bpfioctl Function

Once the device is opened, it is configured with ioctl commands. [Figure 31.12](#) summarizes the ioctl commands used with BPF devices. [Figure 31.13](#) shows the bpfioctl function. Only the code for BIOCSETF and BIOCSETIF is shown. We have omitted the ioctl commands that are not discussed in this text.

**Figure 31.12. BPF ioctl commands.**

Command	Third argument	Function	Description
FIONREAD	u_int	bpfioctl	return #bytes in hold buffer and store buffers.
BIOCGBLEN	u_int	bpfioctl	return size of packet buffers
BIOCSBLEN	u_int	bpfioctl	set size of packet buffers
BIOCSETF	struct bpf_program	bpf_setf	install BPF program
BIOCPLUSH		reset_d	discard pending packets
BIOC PROMISC		ifpromisc	enable promiscuous mode
BIOC GDLT	u_int	bpfioctl	return bif_dlt
BIOC GETIF	struct ifreq	bpf_ifname	return name of attached interface
BIOC SETIF	struct ifreq	bpf_setif	attach network interface to device
BIOC SRTIMEOUT	struct timeval	bpfioctl	set read timeout value
BIOC GRTIMEOUT	struct timeval	bpfioctl	return read timeout value
BIOC GSTATS	struct bpf_stat	bpfioctl	return BPF statistics
BIOC IMMEDIATE	u_int	bpfioctl	enable immediate mode
BIOC VERSION	struct bpf_version	bpfioctl	return BPF version information

**Figure 31.13. bpfioctl function.**

```
----- bpf.c -----
501 int
502 bpfioctl(dev, cmd, addr, flag)
503 dev_t    dev;
504 int     cmd;
505 caddr_t addr;
506 int     flag;
507 {
508     struct bpf_d *d = &bpf_dtab[minor(dev)];
509     int      s, error = 0;
510     switch (cmd) {
511         /*
512          * Set link layer read filter.
513          */
514     case BIOCSETF:
515         error = bpf_setf(d, (struct bpf_program *) addr);
516         break;
517         /*
518          * Set interface.
519          */
520     case BIOCSETIF:
521         error = bpf_setif(d, (struct ifreq *) addr);
522         break;
523
524         /* other ioctl commands from Figure 31.12 */
525
526     default:
527         error = EINVAL;
528         break;
529     }
530     return (error);
531 }
```

----- bpf.c -----

**501-509**

As with bpfopen, the minor device number

selects the bpf\_d structure from the bpf\_dtab table. The command is processed by the cases within the switch. We show two commands, BIOCSETF and BIOCSETIF, as well as the default case.

510-522

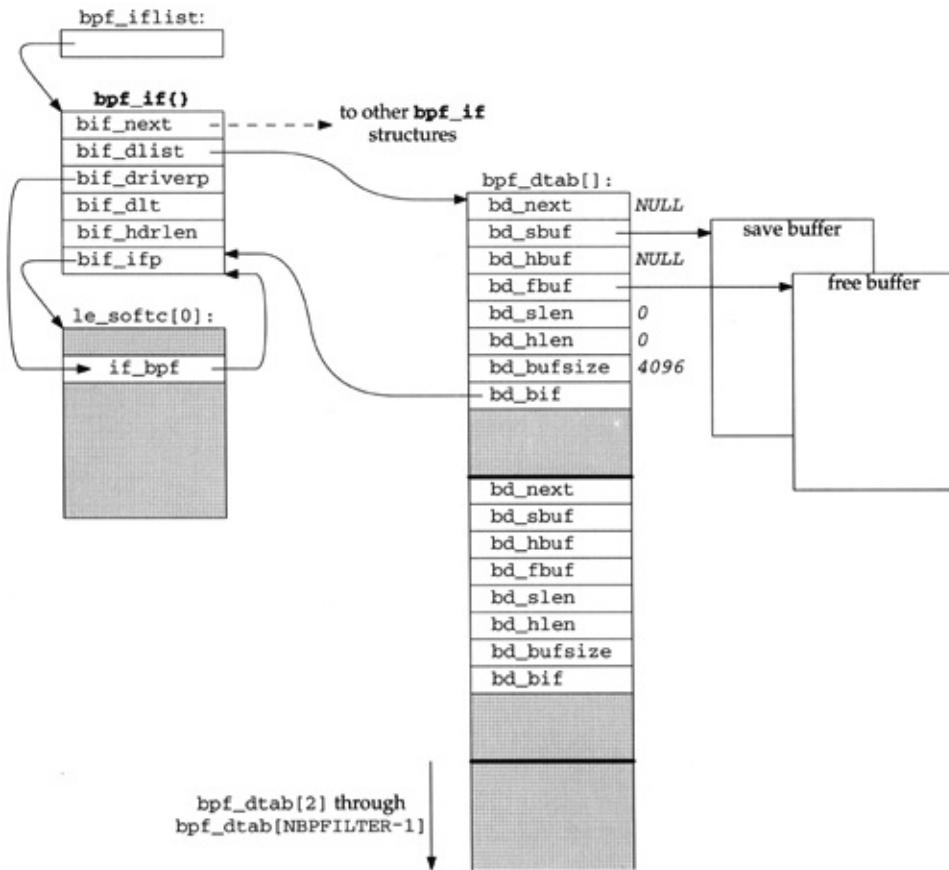
The bpf\_setf function installs the filter passed in addr, and bpf\_setif attaches the named interface to the bpf\_d structure. We don't show the implementation of bpf\_setf in this text.

668-673

If the command is not recognized, EINVAL is returned.

[Figure 31.14](#) shows the bpf\_d structure after bpf\_setif has attached it to the LANCE interface in our example system.

**Figure 31.14. BPF device attached to the Ethernet interface.**



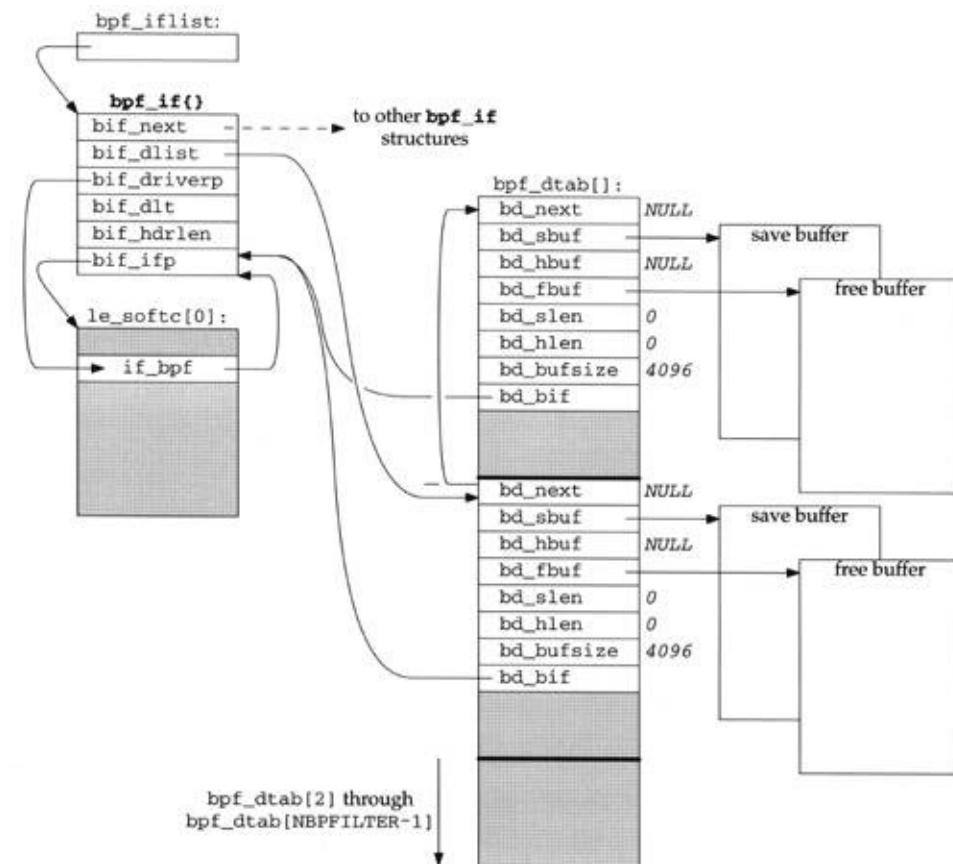
In the figure, `bif_dlist` points to `bpf_dtab[0]`, the first and only descriptor in the descriptor list for the Ethernet interface. In `bpf_dtab[0]`, the `bd_sbuf` and `bd_hbuf` members point to the store and hold buffers. Each buffer is 4096 (`bd_bufsize`) bytes long. `bd_bif` points back to the `bpf_if` structure for the interface.

`if_bpf` in the `ifnet` structure (`le_softc[0]`) also points back to the `bpf_if` structure. As shown in [Figures 4.19](#) and [4.11](#), when `if_bpf` is nonnull, the driver begins passing

packets to the BPF device by calling `bpf_tap`.

Figure 31.15 shows the same structures after a second BPF device is opened and attached to the same Ethernet network interface as in Figure 31.10.

**Figure 31.15. Two BPF devices attached to the Ethernet interface.**



When the second BPF device is opened, a new `bpf_d` structure is allocated from the `bpf_dtab` table, in this case, `bpf_dtab[1]`. The second BPF device is also attached to the Ethernet interface, so `bif_dlist` points to `bpf_dtab[1]`, and `bpf_dtab[1].bd_next` points to `bpf_dtab[0]`, which is the first BPF descriptor attached to the Ethernet interface. Separate store and hold buffers are allocated and attached to the new descriptor structure.

## **bpf\_setif Function**

The `bpf_setif` function, which associates the BPF descriptor with a network interface, is shown in [Figure 31.16](#).

**Figure 31.16. bpf\_setif function.**

```

721 static int
722 bpf_setif(d, ifr)
723 struct bpf_d *d;
724 struct ifreq *ifr;
725 {
726     struct bpf_if *bp;
727     char *cp;
728     int unit, s, error;
729     /*
730      * Separate string into name part and unit number. Put a null
731      * byte at the end of the name part, and compute the number.
732      * If the a unit number is unspecified, the default is 0,
733      * as initialized above. XXX This should be common code.
734      */
735     unit = 0;
736     cp = ifr->ifr_name;
737     cp[sizeof(ifr->ifr_name) - 1] = '\0';
738     while (*cp++) {
739         if (*cp >= '0' && *cp <= '9') {
740             unit = *cp - '0';
741             *cp++ = '\0';
742             while (*cp)
743                 unit = 10 * unit + *cp++ - '0';
744             break;
745         }
746     }
747     /*
748      * Look through attached interfaces for the named one.
749      */
750     for (bp = bpf_iflist; bp != 0; bp = bp->bif_next) {
751         struct ifnet *ifp = bp->bif_ifp;
752
753         if (ifp == 0 || unit != ifp->if_unit
754             || strcmp(ifp->if_name, ifr->ifr_name) != 0)
755             continue;
756         /*
757          * We found the requested interface.
758          * If it's not up, return an error.
759          * Allocate the packet buffers if we need to.
760          * If we're already attached to requested interface,
761          * just flush the buffer.
762          */
763         if ((ifp->if_flags & IFF_UP) == 0)
764             return (ENETDOWN);
765         if (d->bd_sbuf == 0) {
766             error = bpf_allocbufs(d);
767             if (error != 0)
768                 return (error);
769         }
770         s = splimp();
771         if (bp != d->bd_bif) {
772             /*
773              * Detach if attached to something else.
774              */
775             bpf_detachd(d);
776
777             bpf_attachd(d, bp);
778         }
779         reset_d(d);
780         splx(s);
781     }
782     /* Not found. */
783     return (ENXIO);
784 }

```

The first part of bpf\_setif separates the text portion of the name in the ifreq structure ([Figure 4.23](#)) from the numeric portion. The numeric portion is saved in unit. For example, if the first 4 bytes of ifr\_name start is "sl1\0", after this code executes they are "sl\0\0" and unit is 1.

## Locate matching ifnet structure

747-754

The for loop searches the interfaces that support BPF (the ones in bpf\_iflist) for the one specified in the ifreq structure.

755-768

If the matching interface is not up ENETDOWN is returned. If the interface is up, bpf\_allocate attaches the free and store buffers to the bpf\_d structure, if they have not already been allocated.

## Attach bpf\_d structure

769-777

If no interface is attached to the BPF device, or if a different interface from the one specified in the ifreq structure is attached, bpf\_detachd discards the previous interface (if any), and bpf\_attachd attaches the new interface to the device.

778-784

reset\_d resets the packet buffers, discarding any pending packets in the process. The function returns 0 to indicate success or returns ENXIO if the interface was not located.

## bpf\_attachd Function

The bpf\_attachd function shown in [Figure 31.17](#) associates a BPF descriptor structure with a BPF device and with a network interface.

**Figure 31.17. bpf\_attachd function.**

```
189 static void  
190 bpf_attachd(d, bp)  
191 struct bpf_d *d;  
192 struct bpf_if *bp;  
193 {  
194     /*  
195      * Point d at bp, and add d to the interface's list of listeners.  
196      * Finally, point the driver's bpf cookie at the interface so  
197      * it will divert packets to bpf.  
198      */  
199     d->bd_bif = bp;  
200     d->bd_next = bp->bif_dlist;  
201     bp->bif_dlist = d;  
202     *bp->bif_driverp = bp;  
203 }
```

bpf.c

## 189-203

First, bd\_bif is set to point to the BPF interface structure for the network device. Next, the bpf\_d structure is inserted into the front of the list of bpf\_d structures associated with the device. Finally, the BPF pointer within the network interface is changed to point to the BPF structure, which causes the interface to begin passing packets to the BPF device.

## Chapter 31. BPF: BSD Packet Filter

---

### 31.5 BPF Input

Once the BPF device is opened and configured, receive packets from the interface. The BPF tap does not interfere with normal network processing. Instead, it uses the store and hold buffers associated with each interface.

#### bpf\_tap Function

We described the call to bpf\_tap by the LANCE driver in Chapter 29. This call to describe the bpf\_tap. The call (from Figure 31.18) is:

```
bpf_tap(le->sc_if.if_bpf, buf, len);
```

The bpf\_tap function is shown in Figure 31.18.

**Figure 31.18. bpf\_tap Function**

```
869 void  
870 bpf_tap(arg, pkt, pktlen)  
871 caddr_t arg;  
872 u_char *pkt;  
873 u_int  pktlen;  
874 {  
875     struct bpf_if *bp;  
876     struct bpf_d *d;  
877     u_int    slen;  
878     /*  
879      * Note that the ipl does not have to be raised at this point.  
880      * The only problem that could arise here is that if two different  
881      * interfaces shared any data. This is not the case.  
882      */  
883     bp = (struct bpf_if *) arg;  
884     for (d = bp->bif_dlist; d != 0; d = d->bd_next) {  
885         ++d->bd_rcount;  
886         slen = bpf_filter(d->bd_filter, pkt, pktlen, pktlen);  
887         if (slen != 0)  
888             catchpacket(d, pkt, pktlen, slen, bcopy);  
889     }  
890 }
```

— bpf.c

## 869-882

The first argument is a pointer to the bpf\_if structure. The second argument is a pointer to the incoming packet, including the Ethernet header. The third argument is the number of bytes contained in the packet, including the size of the Ethernet header (14 bytes) plus the size of the BPF filter.

## Pass packet to one or more BPF devices

## 883-890

The for loop traverses the list of BPF devices at the head of the bpf\_dlist. The packet is passed to bpf\_filter. If the filter accepts the packet, it is captured and catchpacket saves a copy of the packet. If the slen is 0 and the loop continues. When the loop completes, each BPF device has a separate filter installed with the same network interface.

The loopback driver calls bpf\_mtap to pass pac  
bpf\_tap but copies the packet from an mbuf ch  
memory. This function is not described in this t

## catchpacket Function

In [Figure 31.18](#) we saw that catchpacket is call  
function is shown in [Figure 31.19](#).

**Figure 31.19. catch**

---

```

946 static void
947 catchpacket(d, pkt, pktlen, snaplen, cpfn)
948 struct bpf_d *d;
949 u_char *pkt;
950 u_int pktlen, snaplen;
951 void (*cpfn) (const void *, void *, u_int);
952 {
953     struct bpf_hdr *hp;
954     int totlen, curlen;
955     int hdrlen = d->bd_bif->bif_hdrlen;
956     /*
957      * Figure out how many bytes to move. If the packet is
958      * greater or equal to the snapshot length, transfer that
959      * much. Otherwise, transfer the whole packet (unless
960      * we hit the buffer size limit).
961      */
962     totlen = hdrlen + min(snaplen, pktlen);
963     if (totlen > d->bd_bufsize)
964         totlen = d->bd_bufsize;
965     /*
966      * Round up the end of the previous packet to the next longword.
967      */
968     curlen = BPF_WORDALIGN(d->bd_slen);
969     if (curlen + totlen > d->bd_bufsize) {
970         /*
971          * This packet will overflow the storage buffer.
972          * Rotate the buffers if we can, then wakeup any
973          * pending reads.
974          */
975         if (d->bd_fbuf == 0) {
976             /*
977              * We haven't completed the previous read yet,
978              * so drop the packet.
979              */
980             ++d->bd_dcount;
981             return;
982         }
983         ROTATE_BUFFERS(d);
984         bpf_wakeup(d);
985         curlen = 0;
986     } else if (d->bd_immediate)
987     /*
988      * Immediate mode is set. A packet arrived so any
989      * reads should be woken up.
990      */
991     bpf_wakeup(d);
992     /*
993      * Append the bpf header.
994      */
995     hp = (struct bpf_hdr *) (d->bd_sbuf + curlen);
996     microtime(&hp->bh_tstamp);
997     hp->bh_datalen = pktlen;
998     hp->bh_hdrlen = hdrlen;

999    /*
1000     * Copy the packet data into the store buffer and update its length.
1001     */
1002     (*cpfn) (pkt, (u_char *) hp + hdrlen, (hp->bh_caplen = totlen - hdrlen));
1003     d->bd_slen = curlen + totlen;
1004 }

```

---

**946-955**

The arguments to catchpacket are: d, a pointer pointer to the incoming packet; pktlen the length of bytes to save from the packet; cpfn the packet from pkt to a contiguous area of memory, cpfn is bcopy. When points to the first mbuf in a chain such as with

956-964

In addition to the link-layer header and the payload of the packet. The number of bytes to save from the payload is 964. The resulting packet and bpf\_hdr must fit within

## Will the packet fit?

965-985

curlen is the number of bytes already in the store buffer. If the incoming packet is longer than curlen, the store buffer is full. If a free buffer is available (data from the hold buffer), the incoming packet is rotated into place by ROTATE\_BUFFERS and the process is awakened by bpf\_wakeup.

## Immediate mode processing

986-991

If the device is operating in immediate mode, a process the incoming packet there is no buffer

## Append BPF header

992-1004

The current time (microtime), the packet length bpf\_hdr. The function pointed to by cpfn is called and the length of the store buffer is updated. So even before the packet is transferred from a device, the timestamp is close to the actual reception time.

## bpfread Function

The kernel routes a read on a BPF device to bpf BIOCSRTIMEOUT command. This "feature" is enabled by default in the system call, but tcpdump, for example, uses BIOCGBLEN. The kernel must provide a read buffer that matches the size of the bpfread command returns the size of the buffer. When the store buffer becomes full, the kernel rotates the data copied to the buffer provided with the read system call, collecting incoming packets in the store buffer.

Figure 31.20. bpfread

```
344 int
345 bpfread(dev, uio)
346 dev_t dev;
347 struct uio *uio;
348 {
349     struct bpf_d *d = &bpf_dtab[minor(dev)];
350     int error;
351     int s;
352     /*
353      * Restrict application to use a buffer the same size as
354      * as kernel buffers.
355      */
356     if (uio->uio_resid != d->bd_bufsize)
357         return (EINVAL);
358     s = splimp();
359     /*
360      * If the hold buffer is empty, then do a timed sleep, which
361      * ends when the timeout expires or when enough packets
362      * have arrived to fill the store buffer.
363      */
364     while (d->bd_hbuf == 0) {
365         if (d->bd_immediate && d->bd_slen != 0) {
366             /*
367              * A packet(s) either arrived since the previous
368              * read or arrived while we were asleep.
369              * Rotate the buffers and return what's here.
370              */
371             ROTATE_BUFFERS(d);
372             break;
373         }
374         error = tsleep((caddr_t) d, PRINET | PCATCH, "bpf", d->bd_rtout);
375         if (error == EINTR || error == ERESTART) {
376             splx(s);
377             return (error);
378         }
379         if (error == EWOULDBLOCK) {
380             /*
381              * On a timeout, return what's in the buffer,
382              * which may be nothing. If there is something
383              * in the store buffer, we can rotate the buffers.
384              */
385         if (d->bd_hbuf)
386             /*
387              * We filled up the buffer in between
388              * getting the timeout and arriving
389              * here, so we don't need to rotate.
390              */
391             break;
392     }
393 }
```

```

392         if (d->bd_slen == 0) {
393             splx(s);
394             return (0);
395         }
396         ROTATE_BUFFERS(d);
397         break;
398     }
399 }
400 /*
401 * At this point, we know we have something in the hold slot.
402 */
403 splx(s);

404 /*
405 * Move data from hold buffer into user space.
406 * We know the entire buffer is transferred since
407 * we checked above that the read buffer is bpf_bufsize bytes.
408 */
409 error = uiomove(d->bd_hbuf, d->bd_hlen, UIO_READ, uio);

410 s = splimp();
411 d->bd_fbuf = d->bd_hbuf;
412 d->bd_hbuf = 0;
413 d->bd_hlen = 0;
414 splx(s);

415 return (error);
416 }

```

*bpf.c*

## 344-357

The minor device number selects the BPF device  
doesn't match the size of the BPF device buffer

## Wait for data

## 358-364

Since multiple processes may be reading from the read to continue when some other process hold buffer, the loop is skipped. This is different network interface through two different BPF de

## Immediate mode

365-373

If the device is in immediate mode and there is  
are rotated and the while loop terminates.

## No packets available

374-384

If the device is not in the immediate mode, or the process sleeps until a signal arrives, the read timer expires. If a signal arrives, EINTR or ERESTART is returned by the syscall function and never returned to the process.

Remember that a process never sees the ERESTART error code by the syscall function and never returned to the process.

## Check hold buffer

385-391

If the timer expired and data is in the hold buffer, the read function returns the data.

## Check store buffer

392-399

If the timer expired and there is no data in the store buffer, the read function returns -EAGAIN.

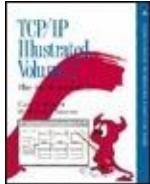
must handle this case when using a timed read store buffer, it is rotated to the hold buffer and

If tsleep returns without an error and data is pr loop terminates.

## Packets are available

400-416

At this point, there is data in the hold buffer. ui the hold buffer to the process. After the move, and the buffer counts are cleared before the fu indicates that uiomove will always be able to cc the read buffer was checked to ensure it can hc bd\_bufsize.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 31. BPF: BSD Packet Filter

### 31.6 BPF Output

Finally, we describe how to add packets to the network interface output queues with BPF. An entire data-link frame must be constructed by the process. For Ethernet this includes the source and destination hardware addresses and the frame type ([Figure 4.8](#)). The kernel will not modify the frame before putting it on the interface's output queue.

#### bpfwrite Function

The frame is passed to the BPF device with the write system call, which the kernel routes to bpfwrite, shown in [Figure 31.21](#).

## Figure 31.21. bpfwrite function.

```
-----bpf.c-----  
437 int  
438 bpfwrite(dev, uio)  
439 dev_t dev;  
440 struct uio *uio;  
441 {  
442     struct bpf_d *d = &bpf_dtab[minor(dev)];  
443     struct ifnet *ifp;  
444     struct mbuf *m;  
445     int error, s;  
446     static struct sockaddr dst;  
447     int datlen;  
448     if (d->bd_bif == 0)  
449         return (ENXIO);  
450     ifp = d->bd_bif->bif_ifp;  
451     if (uio->uio_resid == 0)  
452         return (0);  
453     error = bpf_movein(uio, (int) d->bd_bif->bif_dlt, &m, &dst, &datlen);  
454     if (error)  
455         return (error);  
456     if (datlen > ifp->if_mtu)  
457         return (EMSGSIZE);  
458     s = splnet();  
459     error = (*ifp->if_output) (ifp, m, &dst, (struct rtentry *) 0);  
460     splx(s);  
461     /*  
462      * The driver frees the mbuf.  
463      */  
464     return (error);  
465 }-----bpf.c-----
```

## Check device number

437-449

The minor device number selects the BPF device, which must be attached to a network interface. If it isn't, ENXIO is returned.

## Copy data into mbuf chain

450-457

If the write specified 0 bytes, 0 is returned immediately. `bpf_movein` copies the data from the process into an mbuf chain.

Based on the interface type passed from `bif_dlt`, it computes the length of the packet excluding the link-layer header and returns the value in `datlen`. It also returns an initialized `sockaddr` structure in `dst`. For Ethernet, the type of this address structure will be `af_unspec`, indicating that the mbuf chain contains the data-link header for the outgoing frame. If the packet is larger than the MTU of the interface, `EMSGSIZE` is returned.

## Queue packet

458-465

The resulting mbuf chain is passed to the network interface using the `if_output` function specified in the `ifnet` structure. For Ethernet, `if_output` is `ether_output`.

---

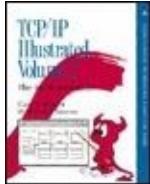
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 31. BPF: BSD Packet Filter

### 31.7 Summary

In this chapter we showed how BPF devices are configured, how incoming frames are passed to BPF devices, and how outgoing frames can be transmitted on a BPF device.

We showed that a single network interface can have multiple BPF taps, each with a separate filter. The store and hold buffers minimize the number of read system calls required to process incoming frames.

We focused only on the major features of BPF in this chapter. For a more detailed description of the filtering code and the other features of the BPF device, the

interested reader should examine the source code and the Net/3 manual pages.

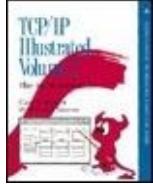
## Exercises

Why is it OK to call `bpf_wakeup` in **31.1** `catchpacket` before the packet is stored in the BPF buffers?

With [Figure 31.20](#), we noted that two processes may be waiting for data from the same BPF device. With **31.2** [Figure 31.11](#), we noted that only one process at a time can open a particular BPF device. How can both of these statements be true?

What happens if the device named in **31.3** the `BIOCSETIF` command does not support BPF?





**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Chapter 32. Raw IP

[Section 32.1. Introduction](#)

[Section 32.2. Code Introduction](#)

[Section 32.3. Raw IP protosw Structure](#)

[Section 32.4. rip\\_init Function](#)

[Section 32.5. rip\\_input Function](#)

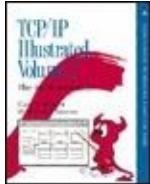
[Section 32.6. rip\\_output Function](#)

[Section 32.7. rip\\_usrreq Function](#)

[Section 32.8. rip\\_ctloutput Function](#)

[Section 32.9. Summary](#)





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.1 Introduction

A process accesses the raw IP layer by creating a socket of type `SOCK_RAW` in the Internet Domain. There are three uses for raw sockets:

- 1. Raw sockets allow a process to send and receive ICMP and IGMP messages.**

**The Ping program uses this type of socket to send ICMP echo requests and to receive ICMP echo replies.**

**Some routing daemons use this feature to track ICMP redirects that are processed by the kernel. We saw in Section 19.7 that Net/3**

**generates an RTM\_REDIRECT message on a routing socket when a redirect is processed, obviating the need for this use of raw sockets.**

**This feature is also used to implement protocols based on ICMP, such as router advertisement and router solicitation (Section 9.6 of Volume 1), which use ICMP but are better implemented as user processes than within the kernel.**

**The multicast routing daemon uses a raw IGMP socket to send and receive IGMP messages.**

- Raw sockets let a process build its own IP headers. The Traceroute program uses this feature to build its own UDP datagrams, including the IP and UDP headers.
- Raw sockets let a process read and write IP datagrams with an IP protocol type that the kernel doesn't support.

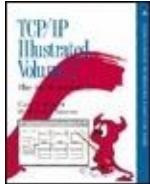
The gated program uses this to support

three routing protocols that are built directly on IP: EGP, HELLO, and OSPF.

This type of raw socket can also be used to experiment with new transport layers on top of IP, instead of adding support to the kernel. It is usually much easier to debug code within a user process than it is within the kernel.

This chapter examines the implementation of raw IP sockets.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.2 Code Introduction

There are five raw IP functions in a single C file, shown in [Figure 32.1](#).

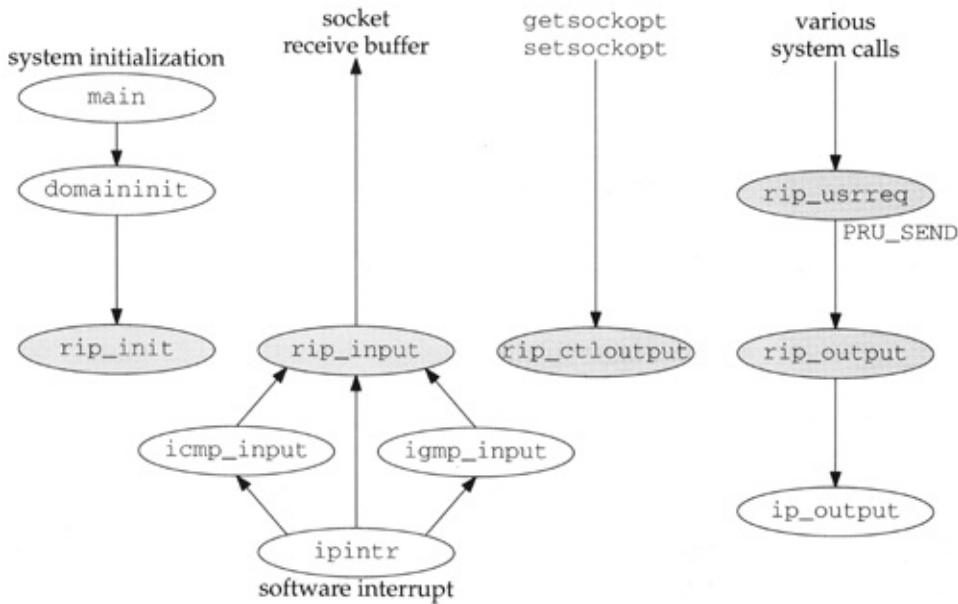
**Figure 32.1. File discussed in this chapter.**

File	Description
netinet/raw_ip.c	raw IP functions

[Figure 32.2](#) shows the relationship of the five raw IP functions to other kernel functions.

**Figure 32.2. Relationship of raw IP**

## functions to rest of kernel.



The shaded ellipses are the five functions that we cover in this chapter. Be aware that the "rip" prefix used within the raw IP functions stands for "raw IP" and not the "Routing Information Protocol," whose common acronym is RIP.

## Global Variables

Four global variables are introduced in this chapter, which are shown in [Figure 32.3](#).

**Figure 32.3. Global variables introduced**

in this chapter.

Variable	Datatype	Description
rawinpcb	struct inpcb	head of the raw IP Internet PCB list
ripsrc	struct sockaddr_in	contains sender's IP address on input
rip_recvspace	u_long	default size of socket receive buffer, 8192 bytes
rip_sendspace	u_long	default size of socket send buffer, 8192 bytes

## Statistics

Raw IP maintains two of the counters in the ipstat structure ([Figure 8.4](#)). We describe these in [Figure 32.4](#).

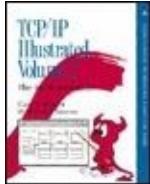
**Figure 32.4. Raw IP statistics maintained in the ipstat structure.**

ipstat member	Description	Used by SNMP
ips_noproto	#packets with an unknown or unsupported protocol	•
ips_rawout	total #raw ip packets generated	

The use of the ips\_noproto counter with SNMP is shown in [Figure 8.6](#). [Figure 8.5](#) shows some sample output of these two counters.

**Team-Fly**

Top



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.3 Raw IP protosw Structure

Unlike all other protocols, raw IP is accessed through multiple entries in the inetsw array. There are four entries in this structure with a socket type of SOCK\_RAW, each with a different protocol value:

- IPPROTO\_ICMP (protocol value of 1),
- IPPROTO\_IGMP (protocol value of 2),
- IPPROTO\_RAW (protocol value of 255), and
- raw wildcard entry (protocol value of 0).

The first two entries for ICMP and IGMP were described earlier ([Figures 11.12](#) and

[13.9](#)). The difference in these four entries can be summarized as follows:

- If the process creates a raw socket (SOCK\_RAW) with a nonzero protocol value (the third argument to socket), and if that value matches IPPROTO\_ICMP, IPPROTO\_IGMP, or IPPROTO\_RAW, then the corresponding protosw entry is used.
- If the process creates a raw socket with a nonzero protocol value that is not known to the kernel, the wildcard entry with a protocol of 0 is matched by pffindproto. This allows a process to handle any IP protocol that is not known to the kernel, without making kernel modifications.

We saw in [Section 7.8](#) that all entries in the ip\_protox array that are unknown are set to point to the entry for IPPROTO\_RAW, whose protocol switch entry we show in [Figure 32.5](#).

**Figure 32.5. The raw IP protosw**

## structure.

Member	inetsw[3]	Description
pr_type	<i>SOCK_RAW</i>	raw socket
pr_domain	<i>&amp;inetdomain</i>	raw IP is part of the Internet domain
pr_protocol	<i>IPPROTO_RAW (255)</i>	appears in the ip_p field of the IP header
pr_flags	<i>PR_ATOMIC/PR_ADDR</i>	socket layer flags, not used by protocol processing
pr_input	<i>rip_input</i>	receives messages from IP layer
pr_output	<i>0</i>	not used by raw IP
pr_ctlinput	<i>0</i>	not used by raw IP
pr_ctloutput	<i>rip_ctloutput</i>	respond to administrative requests from a process
pr_usrreq	<i>rip_usrreq</i>	respond to communication requests from a process
pr_init	<i>0</i>	not used by raw IP
pr_fasttimo	<i>0</i>	not used by raw IP
pr_slowtimo	<i>0</i>	not used by raw IP
pr_drain	<i>0</i>	not used by raw IP
pr_sysctl	<i>0</i>	not used by raw IP

We describe the three functions that begin with `rip_` in this chapter. We also cover the function `rip_output`, which is not in the protocol switch entry but is called by `rip_usrreq` when a raw IP datagram is output.

The fifth raw IP function, `rip_init`, is contained only in the wildcard entry. The initialization function must be called only once, so it could appear in either the `IPPROTO_RAW` entry or in the wildcard entry.

What [Figure 32.5](#) doesn't show, however, is that other protocols (ICMP and IGMP) also reference some of the raw IP

functions in their protosw entries. Figure 32.6 compares the relevant fields in the protosw entries for the four SOCK\_RAW protocols. To highlight the differences, values in these rows are in a bolder font when they differ.

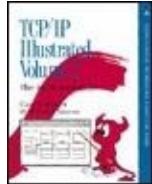
**Figure 32.6. Comparison of protocol switch values for raw sockets.**

protosw entry	SOCK_RAW protocol type			
	IPPROTO_ICMP (1)	IPPROTO_IGMP (2)	IPPROTO_RAW (255)	wildcard (0)
pr_input	<b>icmp_input</b>	<b>igmp_input</b>	<b>rip_input</b>	<b>rip_input</b>
pr_output	<b>rip_output</b>	<b>rip_output</b>	<b>rip_output</b>	<b>rip_output</b>
pr_ctloutput	<b>rip_ctloutput</b>	<b>rip_ctloutput</b>	<b>rip_ctloutput</b>	<b>rip_ctloutput</b>
pr_usrreq	<b>rip_usrreq</b>	<b>rip_usrreq</b>	<b>rip_usrreq</b>	<b>rip_usrreq</b>
pr_init	0	<b>igmp_init</b>	0	0
pr_sysctl	<b>icmp_sysctl</b>	0	0	<b>rip_init</b>
pr_fasttimo	0	<b>igmp_fasttimo</b>	0	0

The implementation of raw sockets has changed with the different BSD releases. The entry with a protocol of IPPROTO\_RAW has always been used as the wildcard entry in the ip\_protox table for unknown IP protocols. The entry with a protocol of 0 has always been the default entry, to allow processes to read and write IP datagrams with a protocol that the kernel doesn't support.

Usage of the IPPROTO\_RAW entry by a process started when Traceroute was developed by Van Jacobson, because Traceroute was the first process that needed to write its own IP headers (to change the TTL field). The kernel patches to 4.3BSD and Net/1 to support Traceroute included a change to rip\_output so that if the protocol was IPPROTO\_RAW, it was assumed the process had passed a complete IP datagram, including the IP header. This was changed with Net/2 when the IP\_HDRINCL socket option was introduced, removing this overloading of the IPPROTO\_RAW protocol and allowing a process to send its own IP header with the wildcard entry.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.4 rip\_init Function

The domaininit function calls the raw IP initialization function rip\_init (Figure 32.7) at system initialization time.

**Figure 32.7. rip\_init function.**

```
47 void  
48 rip_init()  
49 {  
50     rawinpcb.inp_next = rawinpcb.inp_prev = &rawinpcb;  
51 }  
                                         raw_ip.c  
                                         raw_ip.c
```

The only action performed by this function is to set the next and previous pointers in the head PCB (rawinpcb) to point to itself. This is an empty doubly linked list.

Whenever a socket of type SOCK\_RAW is created by the socket system call, we'll see that the raw IP PRU\_ATTACH function creates an Internet PCB and puts it onto the rawinpcb list.

---

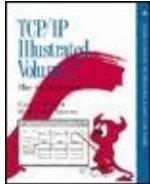
[Team-Fly](#)



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.5 rip\_input Function

Since all entries in the ip\_protox array for unknown protocols are set to point to the entry for IPPROTO\_RAW (Section 7.8), and since the pr\_input function for this protocol is rip\_input (Figure 32.6), this function is called for all IP datagrams that have a protocol value that the kernel doesn't recognize. But from Figure 32.2 we see that both ICMP and IGMP also call rip\_input. This happens under the following conditions:

- icmp\_input calls rip\_input for all unknown ICMP message types and for all ICMP messages that are not reflected.

- igmp\_input calls rip\_input for all IGMP packets.

One reason for calling rip\_input in these two cases is to allow a process with a raw socket to handle new ICMP and IGMP messages that might not be supported by the kernel.

[Figure 32.8](#) shows the rip\_input function.

**Figure 32.8. rip\_input function.**

---

```

59 void
60 rip_input(m)
61 struct mbuf *m;
62 {
63     struct ip *ip = mtod(m, struct ip *);
64     struct inpcb *inp;
65     struct socket *last = 0;
66     ripsrc.sin_addr = ip->ip_src;
67     for (inp = rawinpcb.inp_next; inp != &rawinpcb; inp = inp->inp_next) {
68         if (inp->inp_ip.ip_p && inp->inp_ip.ip_p != ip->ip_p)
69             continue;
70         if (inp->inp_laddr.s_addr &&
71             inp->inp_laddr.s_addr == ip->ip_dst.s_addr)
72             continue;
73         if (inp->inp_faddr.s_addr &&
74             inp->inp_faddr.s_addr == ip->ip_src.s_addr)
75             continue;
76         if (last) {
77             struct mbuf *n;
78             if (n = m_copy(m, 0, (int) M_COPYALL)) {
79                 if (sbappendaddr(&last->so_rcv, &ripsrc,
80                                 n, (struct mbuf *) 0) == 0)
81                     /* should notify about lost packet */
82                     m_freem(n);
83                 else
84                     sorwakeup(last);
85             }
86         }
87         last = inp->inp_socket;
88     }
89     if (last) {
90         if (sbappendaddr(&last->so_rcv, &ripsrc,
91                         m, (struct mbuf *) 0) == 0)
92             m_freem(m);
93         else
94             sorwakeup(last);
95     } else {
96         m_freem(m);
97         ipstat.ips_noproto++;
98         ipstat.ips_delivered--;
99     }
100 }
```

---

— raw\_ip.c

## Save source IP address

59-66

The source address from the IP datagram is put into the global variable ripsrc, which becomes an argument to sbappendaddr whenever a matching PCB is found. Unlike UDP, there is no concept of a port number

with raw IP, so the `sin_port` field in the `sockaddr_in` structure is always 0.

## Search all raw IP PCBs for one or more matching entries

67-88

Raw IP handles its list of PCBs differently from UDP and TCP. We saw that these two protocols maintain a pointer to the PCB for the most recently received datagram (a one-behind cache) and call the generic function `in_pcblklookup` to search for a single "best" match when the received datagram does not equal the cache entry. Raw IP has completely different criteria for a matching PCB, so it searches the PCB list itself. `in_pcblklookup` cannot be used because a raw IP datagram can be delivered to multiple sockets, so every PCB on the raw PCB list must be scanned. This is similar to UDP's handling of a received datagram destined for a broadcast or multicast address ([Figure 23.26](#)).

## Compare protocols

68-69

If the protocol field in the PCB is nonzero, and if it doesn't match the protocol field in the IP header, the PCB is ignored. This implies that a raw socket with a protocol value of 0 (the third argument to socket) can match any received raw IP datagram.

## Compare local and foreign IP addresses

70-75

If the local address in the PCB is nonzero, and if it doesn't match the destination IP address in the IP header, the PCB is ignored. If the foreign address in the PCB is nonzero, and if it doesn't match the source IP address in the IP header, the PCB is ignored.

These three tests imply that a process can create a raw socket with a protocol of 0, not bind a local address, and not connect to a foreign address, and the process receives *all* datagrams processed by rip\_input.

Lines 71 and 74 both contain the same bug: the test for equality should be a test for inequality.

## Pass copy of received datagram to processes

76-94

sbappendaddr passes a copy of the received datagram to the process. The use of the variable last is similar to what we saw in [Figure 23.26](#): since sbappendaddr releases the mbuf after placing it onto the appropriate queue, if more than one process receives a copy of the datagram, rip\_input must make a copy by calling m\_copy. But if only one process receives the datagram, there's no need to make a copy.

## Undeliverable datagram

95-99

If no matching sockets are found for the datagram, the mbuf is released,

`ips_noproto` is incremented, and `ips_delivered` is decremented. This latter counter was incremented by IP just before calling the `rip_input` ([Figure 8.15](#)). It must be decremented so that the two SNMP counters, `ipInDiscards` and `ipInDelivers` ([Figure 8.6](#)) are correct, since the datagram was not really delivered to a transport layer.

At the beginning of this section we mentioned that `icmp_input` calls `rip_input` for unknown message types and for messages that are not reflected. This means that the receipt of an ICMP host unreachable causes `ips_noproto` to be incremented if there are no raw listeners whose PCB is matched by `rip_input`. That's one reason this counter has such a large value in [Figure 8.5](#). The description of this counter as being "unknown or unsupported protocols" is not entirely accurate.

Net/3 does not generate an ICMP destination unreachable message with code 2 (protocol unreachable) when an IP datagram is received with a protocol

field that is not handled by either the kernel or some process through a raw socket. RFC 1122 says an implementation should generate this ICMP error. (See [Exercise 32.4.](#))

---

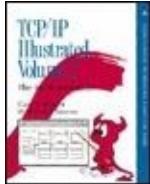
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

## 32.6 rip\_output Function

We saw in [Figure 32.6](#) that `rip_output` is called for output for raw sockets by ICMP, IGMP, and raw IP. Output occurs when the application calls one of the five write functions: `send`, `sendto`, `sendmsg`, `write`, or `writev`. If the socket is connected, any of the five functions can be called, although a destination address cannot be specified with `sendto` or `sendmsg`. If the socket is unconnected, only `sendto` and `sendmsg` can be called, and a destination address must be specified.

The function `rip_output` is shown in [Figure 32.9](#).

## Figure 32.9. rip\_output function.

```
105 int  
106 rip_output(m, so, dst) ————— raw_ip.c  
107 struct mbuf *m;  
108 struct socket *so;  
109 u_long dst;  
110 {  
111     struct ip *ip;  
112     struct inpcb *inp = sotoinpcb(so);  
113     struct mbuf *opts;  
114     int flags = (so->so_options & SO_DONTROUTE) | IP_ALLOWBROADCAST;  
115     /*  
116      * If the user handed us a complete IP packet, use it.  
117      * Otherwise, allocate an mbuf for a header and fill it in.  
118      */  
119     if ((inp->inp_flags & INP_HDRINCL) == 0) {  
120         M_PREPEND(m, sizeof(struct ip), M_WAIT);  
121         ip = mtod(m, struct ip *);  
122         ip->ip_tos = 0;  
123         ip->ip_off = 0;  
124         ip->ip_p = inp->inp_ip.ip_p;  
125         ip->ip_len = m->m_pkthdr.len;  
126         ip->ip_src = inp->inp_laddr;  
127         ip->ip_dst.s_addr = dst;  
128         ip->ip_ttl = MAXTTL;  
129         opts = inp->inp_options;  
130     } else {  
131         ip = mtod(m, struct ip *);  
132         if (ip->ip_id == 0)  
133             ip->ip_id = htons(ip_id++);  
134         opts = NULL;  
135         /* XXX prevent ip_output from overwriting header fields */  
136         flags |= IP_RAWOUTPUT;  
137         ipstat.ips_rawout++;  
138     }  
139     return (ip_output(m, opts, &inp->inp_route, flags, inp->inp_moptions));  
140 }
```

## Kernel fills in IP header

119-128

If the IP\_HDRINCL socket option is not defined, M\_PREPEND allocates room for an IP header, and fields in the IP header are filled in. The fields that are not filled in

here are left for ip\_output to initialize (Figure 8.22). The protocol field is set to the value stored in the PCB, which we'll see in Figure 32.10 is the third argument to the socket system call.

## Figure 32.10. rip\_usrreq function: PRU\_ATTACH request.

```
194 int                                         raw_ip.c
195 rip_usrreq(so, req, m, nam, control)
196 struct socket *so;
197 int     req;
198 struct mbuf *m, *nam, *control;
199 {
200     int     error = 0;
201     struct inpcb *inp = sotoinpcb(so);
202     extern struct socket *ip_mrouter;
203     switch (req) {
204         case PRU_ATTACH:
205             if (inp)
206                 panic("rip_attach");
207             if ((so->so_state & SS_PRIV) == 0) {
208                 error = EACCES;
209                 break;
210             }
211             if ((error = soreserve(so, rip_sendspace, rip_recvspace)) ||
212                 (error = in_pcalloc(so, &rawinpcb)))
213                 break;
214             inp = (struct inpcb *) so->so_pcb;
215             inp->inp_ip.ip_p = (int) nam;
216             break;

```

The TOS is set to 0 and the TTL to 255. These values are always used for a raw socket when the kernel fills in the header. This differs from UDP and TCP where the process had the capability of setting the IP\_TTL and IP\_TOS socket options.

129

Any IP options set by the process with the IP\_OPTIONS socket options are passed to ip\_output through the opts variable.

### **Caller fills in IP header: IP\_HDRINCL socket option**

130-133

If the IP\_HDRINCL socket option is set, the caller supplies a completed IP header at the front of the datagram. The only modification made to this IP header is to set the ID field if the value supplied by the process is 0. The ID field of an IP datagram can be 0. The assignment of the ID field here by rip\_output is just a convention that allows the process to set it to 0, asking the kernel to assign an ID value based on the kernel's current ip\_id variable.

134-136

The opts variable is set to a null pointer, which ignores any IP options the process

may have set with the IP\_OPTIONS socket option. The convention here is that if the caller builds its own IP header, that header includes any IP options the caller might want. The flags variable must also include the IP\_RAWOUTPUT flag, telling ip\_output to leave the header alone.

137

The counter ips\_rawout is incremented. Running Traceroute causes this variable to be incremented by 1 for each datagram sent by Traceroute.

The operation of rip\_output has changed over time. When the IP\_HDRINCL socket option is used in Net/3, the only change made to the IP header by rip\_output is to set the ID field, if the process sets it to 0. The Net/3 ip\_output function does nothing to the IP header fields because the IP\_RAWOUTPUT flag is set. Net/2, however, always set certain fields in the IP header, even if the IP\_HDRINCL socket option was set: the IP version was set to 4, the fragment offset was

set to 0, and the more-fragments flag was cleared.

---

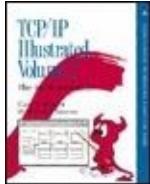
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.7 rip\_usrreq Function

The protocol's user-request function is called for a variety of operations. As with the UDP and TCP user-request functions, rip\_usrreq is a large switch statement, with one case for each PRU\_xxx request.

The PRU\_ATTACH request, shown in [Figure 32.10](#), is from the socket system call.

194-206

Since the socket function creates a new socket structure each time it is called, that structure cannot point to an Internet PCB.

**Verify superuser**

207-210

Only the superuser can create a raw socket. This is to prevent random users from writing their own IP datagrams to the network.

## Create Internet PCB and reserve buffer space

211-215

Space is reserved for input and output queues, and in\_pcalloc allocates a new Internet PCB. The PCB is added to the raw IP PCB list (rawinpcb). The PCB is linked to the socket structure. The nam argument to rip\_usrreq is the third argument to the socket system call: the protocol. It is stored in the PCB since it is used by rip\_input to demultiplex received datagrams, and its value is placed into the protocol field of outgoing datagrams by rip\_output (if IP\_HDRINCL is not set).

A raw IP socket can be connected to a foreign IP address similar to a UDP socket

being connected to a foreign IP address. This fixes the foreign IP address from which the raw socket receives datagrams, as we saw in `rip_input`. Since raw IP is a connectionless protocol like UDP, a `PRU_DISCONNECT` request can occur in two cases:

- 1. When a connected raw socket is closed, `PRU_DISCONNECT` is called before `PRU_DETACH`.**

- When a connect is issued on an already-connected raw socket, `soconnect` issues the `PRU_DISCONNECT` request before the `PRU_CONNECT` request.

[Figure 32.11](#) shows the `PRU_DISCONNECT`, `PRU_ABORT`, and `PRU_DETACH` requests.

**Figure 32.11. `rip_usrreq` function:  
`PRU_DISCONNECT`, `PRU_ABORT`, and  
`PRU_DETACH` requests.**

```
217     case PRU_DISCONNECT:
218         if ((so->so_state & SS_ISCONNECTED) == 0) {
219             error = ENOTCONN;
220             break;
221         }
222         /* FALLTHROUGH */
223     case PRU_ABORT:
224         soisdisconnected(so);
225         /* FALLTHROUGH */
226     case PRU_DETACH:
227         if (inp == 0)
228             panic("rip_detach");
229         if (so == ip_mrouter)
230             ip_mrouter_done();
231         in_pcbdetach(inp);
232         break;
```

raw\_ip.c

## 217-222

The socket must already be connected to disconnect or else an error is returned.

## 223-225

A PRU\_ABORT abort should never be issued for a raw IP socket, but this case also handles the fall through from PRU\_DISCONNECT. The socket is marked as disconnected.

## 226-230

The close system call issues the PRU\_DETACH request, and this case also handles the fall through from the PRU\_DISCONNECT request. If the socket structure is the one used for multicast

routing (`ip_mrouner`), multicast routing is disabled by calling `ip_mrouted_done`. Normally the `mrouted(8)` daemon issues the `DVMRP_DONE` socket option to disable multicast routing, so this check handles the case of the router daemon terminating (i.e., crashing) without issuing the socket option.

231

The Internet PCB is released by `in_pcbsdetach`, which also removes the PCB from the list of raw IP PCBs (`rawinpcb`).

A raw IP socket can be bound to a local IP address with the `PRU_BIND` request, shown in [Figure 32.12](#). We saw in `rip_input` that the socket will receive only datagrams sent to this IP address.

**Figure 32.12. `rip_usrreq` function:  
`PRU_BIND` request.**

```

233     case PRU_BIND:
234     {
235         struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);
236
237         if (nam->m_len != sizeof(*addr)) {
238             error = EINVAL;
239             break;
240         }
241         if ((ifnet == 0) ||
242             ((addr->sin_family != AF_INET) &&
243              (addr->sin_family != AF_IMPLINK)) ||
244             (addr->sin_addr.s_addr &&
245              ifa_ifwithaddr((struct sockaddr *) addr) == 0)) {
246             error = EADDRNOTAVAIL;
247             break;
248         }
249         inp->inp_laddr = addr->sin_addr;
250     }

```

raw\_ip.c

## 233-250

The process fills in a `sockaddr_in` structure with the local IP address. The following three conditions must all be true, or else the error `EADDRNOTAVAIL` is returned:

### **1. at least one interface must be configured,**

- the address family must be `AF_INET` (or `AF_IMPLINK`, a historical artifact), and
- if the IP address being bound is not `0.0.0.0`, it must correspond to a local interface. For the call to `ifa_ifwithaddr` to succeed, the port number in the caller's `sockaddr_in` must be 0.

The local IP address is stored in the PCB.

A process can also connect a raw IP socket to a particular foreign IP address. We saw in rip\_input that this restricts the process so that it receives only IP datagrams with a source IP address equal to the connected IP address. A process has the option of calling bind, connect, both, or neither, depending on the type of filtering it wants rip\_input to place on received datagrams. Figure 32.13 shows the PRU\_CONNECT request.

### Figure 32.13. rip\_usrreq function: PRU\_CONNECT request.

```
251     case PRU_CONNECT:
252     {
253         struct sockaddr_in *addr = mtod(nam, struct sockaddr_in *);
254
255         if (nam->m_len != sizeof(*addr)) {
256             error = EINVAL;
257             break;
258         }
259         if (ifnet == 0) {
260             error = EADDRNOTAVAIL;
261             break;
262         }
263         if ((addr->sin_family != AF_INET) &&
264             (addr->sin_family != AF_IMPLINK)) {
265             error = EAFNOSUPPORT;
266             break;
267         }
268         inp->inp_faddr = addr->sin_addr;
269         soisconnected(so);
270     }
```

If the caller's sockaddr\_in is initialized correctly and at least one IP interface is configured, the specified foreign IP address is stored in the PCB. Notice that this process differs from the connection of a UDP socket to a foreign address. In the UDP case, in\_pcbconnect acquires a route to the foreign address and also stores the outgoing interface as the local address ([Figure 22.9](#)). With raw IP, only the foreign IP address is stored in the PCB, and unless the process also calls bind, only the foreign address is compared by rip\_input.

A call to shutdown specifying that the process has finished sending data generates the PRU\_SHUTDOWN request, although it is rare for a process to issue this system call for a raw IP socket. [Figure 32.14](#) shows the PRU\_CONNECT2 and PRU\_SHUTDOWN requests.

**Figure 32.14. rip\_usrreq function:  
PRU\_CONNECT2 and PRU\_SHUTDOWN  
requests.**

```
271     case PRU_CONNECT2:  
272         error = EOPNOTSUPP;  
273         break;  
  
274     /*  
275      * Mark the connection as being incapable of further input.  
276      */  
277     case PRU_SHUTDOWN:  
278         socantsendmore(so);  
279         break;
```

raw\_ip.c

271-273

The PRU\_CONNECT2 request is not supported for a raw IP socket.

274-279

socantsendmore sets the socket's flags to prevent any future output.

In [Figure 23.14](#) we showed how the five write functions call the protocol's pr\_usrreq function with a PRU\_SEND request. We show this request in [Figure 32.15](#).

**Figure 32.15. rip\_usrreq function:  
PRU\_SEND request.**

```
280     /*  
281      * Ship a packet out.  The appropriate raw output  
282      * routine handles any massaging necessary.  
283      */  
284     case PRU_SEND:  
285     {  
286         u_long dst;  
287  
288         if (so->so_state & SS_ISCONNECTED) {  
289             if (nam) {  
290                 error = EISCONN;  
291                 break;  
292             }  
293             dst = inp->inp_faddr.s_addr;  
294         } else {  
295             if (nam == NULL) {  
296                 error = ENOTCONN;  
297                 break;  
298             }  
299             dst = mtod(nam, struct sockaddr_in *)->sin_addr.s_addr;  
300         }  
301         error = rip_output(m, so, dst);  
302         m = NULL;  
303         break;  
304     }  
----- raw_ip.c
```

## 280-303

If the socket state is connected, the caller cannot specify a destination address (the nam argument). Likewise, if the state is unconnected, a destination address is required. If all is OK, in either state, dst is set to the destination IP address.

rip\_output sends the datagram. The mbuf pointer m is set to a null pointer, to prevent it from being released at the end of the function. This is because the interface output routine will release the mbuf after it has been output. (Remember that rip\_output passes the mbuf chain to ip\_output, who appends it to the interface's output queue.)

The final part of rip\_usrreq is shown in [Figure 32.16](#). The PRU\_SENSE request, generated by the fstat system call, returns nothing. The PRU\_SOCKADDR and PRU\_PEERADDR requests are from the getsockname and getpeername system calls, respectively. The remaining requests are not supported.

## Figure 32.16. rip\_usrreq function: remaining requests.

```
304     case PRU_SENSE:
305         /*
306          * fstat: don't bother with a blocksize.
307          */
308         return (0);
309
310         /*
311          * Not supported.
312          */
313     case PRU_RCVOOB:
314     case PRU_RCVD:
315     case PRU_LISTEN:
316     case PRU_ACCEPT:
317     case PRU_SENDOOB:
318         error = EOPNOTSUPP;
319         break;
320
321     case PRU_SOCKADDR:
322         in_setsockaddr(inp, nam);
323         break;
324
325     case PRU_PEERADDR:
326         in_setpeeraddr(inp, nam);
327         break;
328
329     default:
330         panic("rip_usrreq");
331     }
332     if (m != NULL)
333         m_freem(m);
334     return (error);
335 }
```

The functions `in_setsockaddr` and `in_setpeeraddr` fetch the information from the PCB, storing the result in the `nam` argument.

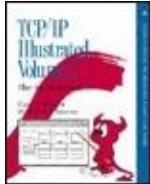
---

**Team-Fly**

[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.8 rip\_ctloutput Function

The setsockopt and getsockopt system calls invoke the rip\_ctloutput function. Only one IP socket option is handled here, along with eight socket options related to multicast routing.

Figure 32.17 shows the first part of the rip\_ctloutput function.

**Figure 32.17. rip\_usrreq function:  
process IP\_HDRINCL socket option.**

---

```

144 int
145 rip_ctloutput(op, so, level, optname, m)
146 int     op;
147 struct socket *so;
148 int     level, optname;
149 struct mbuf **m;
150 {
151     struct inpcb *inp = sotoinpcb(so);
152     int     error;
153
154     if (level != IPPROTO_IP)
155         return (EINVAL);
156
157     switch (optname) {
158
159     case IP_HDRINCL:
160         if (op == PRCO_SETOPT || op == PRCO_GETOPT) {
161             if (m == 0 || *m == 0 || (*m)->m_len < sizeof(int))
162                 return (EINVAL);
163             if (op == PRCO_SETOPT) {
164                 if (*mtod(*m, int *) )
165                     inp->inp_flags |= INP_HDRINCL;
166                 else
167                     inp->inp_flags &= ~INP_HDRINCL;
168                 (void) m_free(*m);
169             }
170             return (0);
171         }
172     break;

```

---

*raw\_ip.c*

## 144-172

The size of the mbuf that contains either the new value of the option or will hold the current value of the option must be at least as large as an integer. For the setsockopt system call, the flag is set if the integer value in the mbuf is nonzero, or cleared otherwise. For the getsockopt system call, the value returned in the mbuf is either 0 or the nonzero value of the flag. The function returns, to avoid the processing at the end of the switch statement for other IP options.

[Figure 32.18](#) shows the last portion of the rip\_ctloutput function. It handles eight multicast routing socket options.

### **Figure 32.18. rip\_usrreq function: process multicast routing socket option.**

```
173     case DVMRP_INIT:
174     case DVMRP_DONE:
175     case DVMRP_ADD_VIF:
176     case DVMRP_DEL_VIF:
177     case DVMRP_ADD_LGRP:
178     case DVMRP_DEL_LGRP:
179     case DVMRP_ADD_MRT:
180     case DVMRP_DEL_MRT:
181
182         /* shown in Figure 14.9 */
183
184     }
185     return (ip_ctloutput(op, so, level, optname, m));
186 }
187
188 }
```

173-188

These eight socket options are valid only for the setsockopt system call. They are processed by the ip\_mrouter\_cmd function as discussed with [Figure 14.9](#).

189

Any other IP socket options, such as IP\_OPTIONS to set the IP options, are processed by ip\_ctloutput.

---

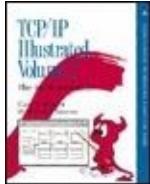
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Chapter 32. Raw IP

### 32.9 Summary

Raw sockets provide three capabilities for an IP host.

#### **1. They are used to send and receive ICMP and IGMP messages.**

- They allow a process to build its own IP headers.
- They allow additional IP-based protocols to be supported in a user process.

We saw that raw IP output is simple—it just fills in a few fields in the IP header—but it allows a process to supply its own IP header. This allows diagnostic programs to create any type of IP datagram.

Raw IP input provides three types of filtering for incoming IP datagrams. The process chooses to receive datagrams based on (1) the protocol field, (2) the source IP address (set by connect), and (3) the destination IP address (set by bind). The process chooses which combination of these three filters (if any) to apply.

## Exercises

**32.1** Assume the IP\_HDRINCL socket option is not set. What value will rip\_output place into the IP header protocol field (ip\_p) when the third argument to socket is 0? What value will rip\_output place into this field when the third argument to socket is IPPROTO\_RAW(255)?

**32.2** A process creates a raw socket with a protocol value of IPPROTO\_RAW (255). What type of IP datagrams will the process receive on this

socket?

A process creates a raw socket with  
**32.3** a protocol value of 0. What type of IP  
datagrams will the process receive  
on this socket?

Modify `rip_input` to send an ICMP  
destination unreachable with code 2  
(protocol unreachable) when  
**32.4** appropriate. Be careful not to  
generate the error for received ICMP  
and IGMP packets for which `rip_input`  
is called.

If a process wants to write its own IP  
datagrams with its own IP header,  
**32.5** what are the differences in using a  
raw IP socket with the `IP_HDRINCL`  
option, and using BPF ([Chapter 31](#))?

When would a process read from a  
**32.6** raw IP socket, and when would it  
read from BPF?

---

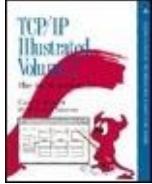
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Epilogue

*"We have come a long way. Nine chapters stuffed with code is a lot to negotiate. If you didn't assimilate all of it the first time through, don't worry you weren't really expected to. Even the best of code takes time to absorb, and you seldom grasp all the implications until you try to use and modify the program. Much of what you learn about programming comes only from working with the code: reading, revising and rereading."*

From the Epilogue of *Software Tools* [Kernighan and Plauger 1976].

*"In fact, this RFC will argue that modularity is one of the chief villains in attempting to obtain good performance, so that the designer is faced with a delicate and inevitable tradeoff between good structure and good performance."*

From RFC 817 [Clark 1982].

This text has provided a long and detailed examination of a significant piece of a real operating system. Versions of the code presented in the text are shipped as part of the Unix kernel with most flavors of Unix today, along with many non-Unix systems.

The code that we've examined is not perfect and it is not the only way to write a TCP/IP protocol stack. It has been modified, enhanced, tweaked, and maligned over the past 15 years by many people. Large portions of the code that we've presented weren't even written at the U. C. Berkeley Computer Systems Research Group: the multicasting code was written by Steve Deering, the long fat pipe support was added by Thomas Skibo, portions of the TCP code were written by Van Jacobson, and so on. The code contains gotos (221 to be exact), many large functions (e.g., `tcp_input` and `tcp_output`), and numerous examples of questionable coding style. (We tried to note these items when discussing the code.) Nevertheless, the code is unquestionably "industrial strength" and

continues to be the base upon which new features are added and the standard upon which other implementations are measured.

The Berkeley networking code was designed on VAXes when a VAX-11/780 with 4 megabytes of memory was a big system. For that reason some of the design features (e.g., mbufs) emphasized memory savings over higher performance. This would change if the code were rewritten from scratch today.

There has been a strong push over the last few years toward higher performance of networking software, as the underlying networks become faster (e.g., FDDI and ATM) and as high-bandwidth applications become more prevalent (e.g., voice and video). Whenever designing networking software within the kernel of an operating system, clarity normally gives way to speed [[Clark 1982](#)]. This will continue in any real-world implementation.

The research implementation of the Internet protocols described in [[Partridge](#)

[Jacobson 1993] and [Jacobson 1993] is a move toward much higher performance.

[Jacobson 1993] reports the code is 10 to 100 times faster than the implementation described in this book. Mbufs, software interrupts, and much of the protocol layering evident in BSD systems are gone. If widely released, this implementation could become the standard that others are measured against in the future.

In July 1994 the successor to IP version 4, IP version 6 (IPv6), was announced. It uses 128-bit (16-byte) addresses. Many changes will take place with the IP and ICMP protocols, but the transport layers, UDP and TCP, will remain virtually the same. (There is talk of a TCPng, the next generation of TCP, but the authors think just upgrading IP will provide enough of a challenge for the hundreds of vendors and millions of users across the world to put off any changes to TCP.) It will take a year or two for vendor-supported implementations to appear, and many years after that for end users to migrate their hosts and routers to IPv6. Research implementations of IPv6 based on the

code in this text should appear in early 1995.

To continue your understanding of the Berkeley networking code, the best course of action at this point is to obtain the source code, and modify it. The source code is easily obtainable ([Appendix B](#)) and numerous exercises throughout the text suggest modifications.

---

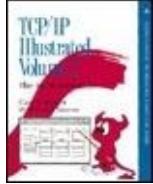
[Team-Fly](#)



[Previous](#)

[Next](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# **Appendix A. Solutions to Selected Exercises**

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[Chapter 12](#)

Chapter 13

Chapter 14

Chapter 15

Chapter 16

Chapter 17

Chapter 18

Chapter 19

Chapter 20

Chapter 21

Chapter 22

Chapter 23

Chapter 24

Chapter 25

Chapter 26

Chapter 27

[Chapter 28](#)

[Chapter 29](#)

[Chapter 30](#)

[Chapter 31](#)

[Chapter 32](#)

---

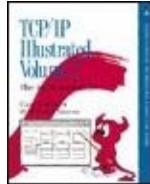
**Team-Fly**



[Previous](#)

[Next](#)

[Top](#)

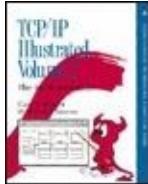


[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 1

SLIP drivers execute at spltty ([Figure 1.13](#)), which must be a priority lower **1.2** than or equal to splimp and must be a priority higher than splnet. Therefore the SLIP drivers are blocked from interrupting.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 2

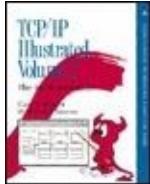
**2.1** The M\_EXT flag is a property of the mbuf itself, not a property of the packet described by the mbuf.

**2.2** The caller asks for more than 100 (MHLEN) contiguous bytes.

This is infeasible since clusters can be pointed to by multiple mbufs ([Section 2.9](#)). Also, there is no room in a

## **2.3** cluster for a back pointer ([Exercise 2.4](#)).

In the macros MCLALLOC and MCLFREE in <sys/mbuf.h> we see that the reference count is an array **2.4** named mclrefcnt. This array is allocated when the kernel is initialized in the file machdep.c.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

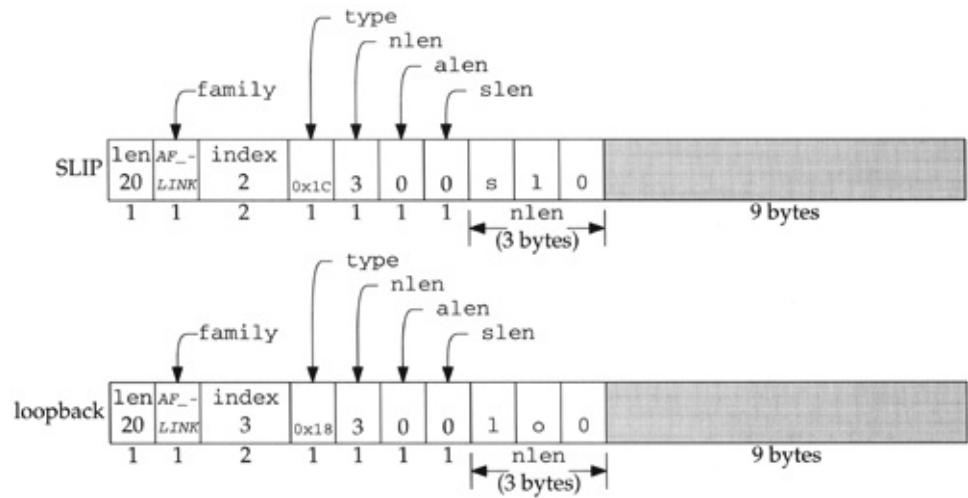
## Appendix A. Solutions to Selected Exercises

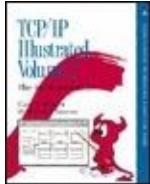
# Chapter 3

**3.3** A large interactive queue would defeat the purpose of the queue by delaying new interactive traffic behind the existing interactive data.

**3.4** Since the `sl_softc` structures are all declared as global variables, they are initialized to 0 when the kernel starts.

### 3.5





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 4

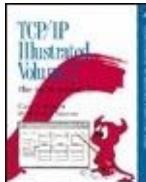
The reader must examine the packet to decide if it needs to be discarded after it is passed to BPF. Since a BPF tap can enable promiscuous mode on the interface, packets may be addressed to some other system on the Ethernet and must be discarded after BPF has processed them.

When the interface is not tapped, the tests must be done in ether\_input.

If the tests were reversed, the broadcast flag would never be set.

**4.2** If the second if wasn't preceded by an else, every broadcast packet would also have the multicast flag set.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 5

**5.1** The loopback interface does not need an input function because all its packets are received directly from looutput, which performs the "input" functions.

**5.2** The stack allocation is faster than dynamic memory allocation. Performance is important for BPF processing, since the code is executed for each incoming packet.

The first character that overflows the buffer is discarded, SC\_ERROR is set, and slinput resets the cluster pointers to begin **5.5** collecting characters at the start of the buffer. Because SC\_ERROR is set, slinput discards the frame when it receives the SLIP END character.

IP discards the packet when the **5.6** checksum is found to be invalid or when it notices that the length in the IP header does not match the physical packet size.

Since ifp points to the first member of a le\_softc structure,

**5.7** `sc = (struct le_softc *) ifp;`

initializes sc correctly.

This is very hard to do. Some routers may send ICMP source quench messages when they begin discarding packets but Net/3 discards these messages for UDP sockets (Figure 23.30). An application would have to begin using the same techniques used by TCP: estimation of the available bandwidth and delay on roundtrip times for acknowledged datagrams.

---

## Appendix A. Solutions to Selected Exercises

---

# Chapter 6

Before IP subnetting (RFC 950 [Mogul and Postel 1985]), IP addresses always appeared on byte boundaries.

**6.1**

```
struct in_addr {
    union {
        struct { u_char s_un_b1; } S_un_b1;
        struct { u_short s_un_w2; } S_un_w2;
        u_long S_addr;
    } S_un;
#define s_addr S_un.S_addr
#define s_host S_un.S_un_b.s_un_b1
#define s_net S_un.S_un_b.s_un_w2
#define s_imp S_un.S_un_w.s_un_b1
#define s_imph S_un.S_un_b.s_un_w.s_un_b1
```

```
#define s_lh      S_un.S_un_b.s  
};
```

The Internet address could be accessed as The macros s\_host, s\_net, s\_imp, and so on structure of early TCP/IP networks.

The use of subnetting and supernetting makes

**6.2** A pointer to the structure labeled sl\_softc[0]

**6.3** The interface output functions, such as eth\_output(), take a pointer to a structure labeled sl\_softc[0] for the interface, and not to an ifaddr structure (which is the last IP address assigned to the interface) or to the ifaddr address list.

**6.4** Only a superuser process can create a raw socket. Raw sockets can examine the interface configurations but they cannot modify the interface addresses.

Three functions loop through a netmask 11111111 11111111 11111111 11111111

## **6.5** ifaof\_ifpforaddr, and rt\_maskedcopy. A short summary of these functions.

The Telnet connection is established with the **6.6** these packets, and other systems should not interface other than the loopback interface.

---

[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

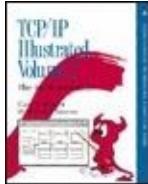
## Appendix A. Solutions to Selected Exercises

# Chapter 7

The following call returns a pointer to inets

### 7.1

`pffindproto(PF_INET, 0, SOCK_`



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 8

**8.1** Probably not. The system could not respond to any broadcasts since it would have no source address to use in the reply.

**8.4** Since the packet has been damaged, there is no way of knowing if the addresses in the header are correct or not.

If an application selects a source address that differs from the address of the selected outgoing interface, redirects from the selected next-hop router fail. The next-hop router sees a source address different from that of the subnetwork on which it was transmitted and does not send a redirect message. This is a consequence of implementing the weak end system model and is noted in RFC 1122.

The new host thinks the broadcast packet is the address of some other host in the unsubnetted network and tries to send it back out on the network. The network interface begins broadcasting ARP requests for the broadcast address, which are never answered.

The decrement of the TTL is done **8.7** after the comparison for less than or equal to 1 to avoid the potential error of decrementing a received TTL of 0 to become 255.

If two routers each consider the other the best next-hop for a packet, a routing loop exists. Until the loop is removed, the original packet bounces between the two routers and each one sends an ICMP redirect back to the source host if that host is on the same network as the routers.

**8.8** Loops may exist when the routing tables are temporarily inconsistent during a routing update.

The TTL of the original packet eventually reaches 0 and the packet is discarded. This is one of the primary reasons why the TTL field exists.

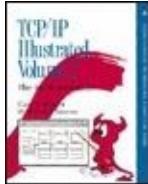
The four Ethernet broadcast addresses would not be checked because they do not belong to the receiving interface. The limited-

**8.9** broadcast addresses would be checked. This implies that a system on a SLIP link can communicate with the system on the other end without knowing the other system's address by utilizing the limited-broadcast address.

ICMP error messages are generated only for the initial fragment of a

**8.10** datagram, which always has an offset of 0. The host and network forms for 0 are the same, so no conversion is necessary.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 9

RFC 1122 says that the behavior is implementation dependent when conflicting options appear in a packet.

- 9.1** Net/3 processes the first source route option correctly, but since this updates ip\_dst in the packet header, the second source route processing will be incorrect.

The host within the network can be used as a relay to access other hosts

within the network. To communicate with an otherwise-blocked host, the source host need only construct packets with a loose route to the relay host and then to the final destination host. The router does not drop the packets because the destination address is the relay host, which will process the route and forward the packet to the final destination host. The destination host reverses the route and uses the relay host to return packets.

The same principle from the previous exercise applies. We pick a relay router that can communicate with the source and destination hosts and construct source routes to pass through the relay and to the destination. The relay router must be on the same network as the destination host so that a default route is not required for communication.

This technique can be extended to allow two hosts to communicate even if they do not have routes to each other, as long as they can find willing relay hosts.

#### **9.4**

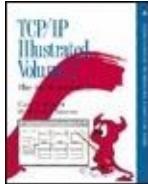
If the source route is the only IP option, the NOP option causes all the IP addresses to be on a 4-byte boundary in the IP header. This can optimize memory references to these addresses on many architectures. This alignment technique also works when multiple options are present if each option is padded with NOPs to a 4-byte boundary.

#### **9.5**

A nonstandard time value cannot be confused with a standard value since the largest standard time value is 86,399,999 (24x60x60x10001) and this value can be represented in 28 bits, which avoids any conflict with the high-order bit since time values are 32

bits long.

The source route option code may change ip\_dst in the packet during **9.6** processing. The destination is saved so that the timestamp processing code uses the original destination.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 10

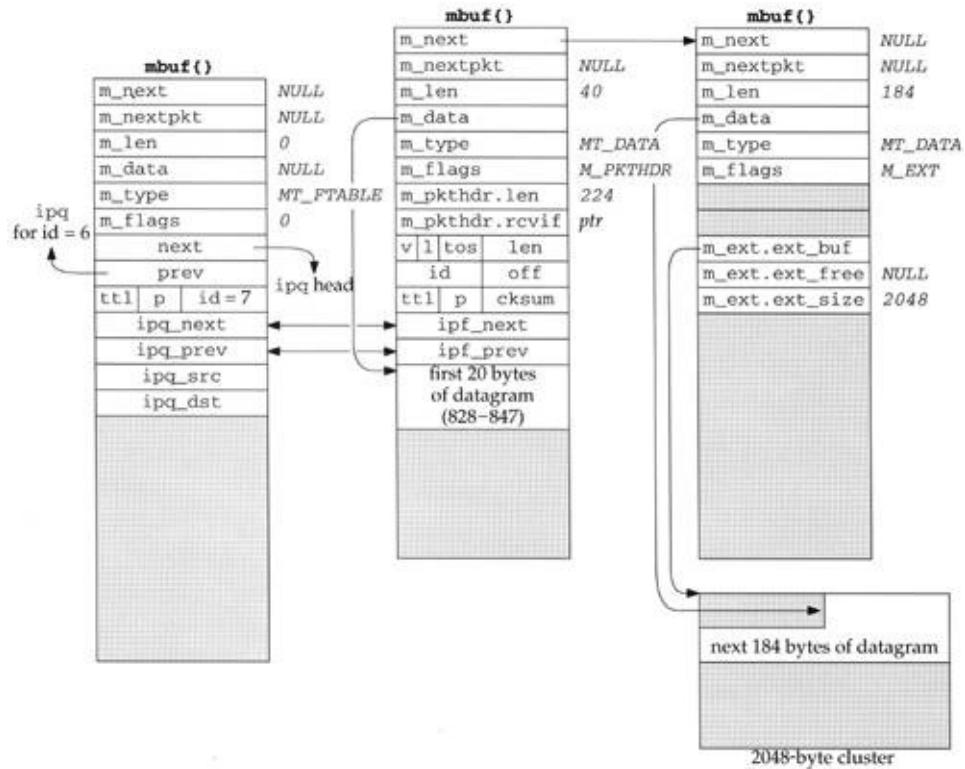
After reassembly, only the options **10.2** from the initial fragment are available to the transport protocols.

The fragment is read into a cluster since the data length ( $216 + 20$ ) is greater than 208 ([Figure 2.16](#)).

`m_pullup` in [Figure 10.11](#) moves the first 40 bytes into a separate mbuf

as in Figure 2.18.

## 10.3



The average number of received fragments per datagram is

$$\frac{72,786 - 349}{16,557} = 4.4$$

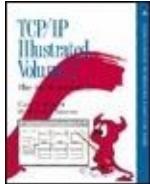
## 10.5

The average number of fragments

created for an outgoing datagram is

$$\frac{796,084}{260,484} = 3.1$$

In Figure 10.11 the packet is initially processed as a fragment. The reserved bit is discarded when ip\_off **10.6** is left shifted. The resulting packet is processed as a fragment or as a complete datagram, depending on the values of the MF and offset bits.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 11

**11.1** The outgoing reply uses the source address of the interface on which the request was received. Hosts are not required to recognize 0.0.0.0 as a valid broadcast address, so the request may be ignored. The recommended broadcast address is 255.255.255.255.

Assume that a host sends link-level broadcasts packets with the IP

source address of another host and the packet contains errors such as an improperly formed option. Every host receives and detects the error because of the link-level broadcast and because options are processed before a final destination check.

**11.2** Many hosts that detect the error try to send an ICMP message back to the IP source of the packet even though the original packet was sent as a link-level broadcast. The unfortunate host will begin receiving many bogus ICMP error messages. This is one reason why ICMP errors must not be sent in response to link-level broadcasts.

In the first case, such a redirect message can fool the host into sending packets to an arbitrary host on an alternate subnetwork. This host may be masquerading as a router but recording the traffic it receives instead. RFC 1009 requires that routers only generate redirect

messages for other routers on the same subnet. Even if the host **11.3** ignores these messages to redirect packets to a new subnetwork, a host on the same subnetwork can fool the host. The second case guards against this by requiring that the host only accept the redirect advice from the original router that it had (erroneously) selected to receive the traffic. Presumably this incorrect router was a default router specified by an administrator.

By passing the message to `rip_input`, a process-level daemon **11.4** could respond and old systems that relied on this behavior could continue to be supported.

ICMP errors are sent only for the initial fragment of an IP datagram. **11.5** Since the offset value of an initial

fragment is always 0, the byte ordering of the field is unimportant.

If the ICMP request was received on an interface that was not yet **11.6** configured with an IP address, ia would be null and no reply could be generated.

**11.7** Net/3 reflects the data along with the timestamp reply.

The high-order bit is reserved and **11.10** must be 0. If it is sent, icmp\_error will discard the packet.

The return value is discarded because icmp\_send does not return

**11.11** an error, but more significantly, errors generated during ICMP processing are discarded to avoid generating an endless series of error messages.

---

[TCP/IP Illustrated, Volume 2: The Implementation](#) By Ga  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 12

On an Ethernet, the IP broadcast address 255.255.255.255 translates to the Ethernet address ff:ff:ff:ff:ff:ff and is received by all interfaces on the network. Systems that have software must actively receive and discard broadcast packets.

**12.1** A packet sent to the IP all-hosts multicast address 224.0.0.1 translates to the Ethernet multicast address 01:00:5e:00:00:01 and is received only by systems that have explicitly instructed their interfaces to receive multicast datagrams. Systems that aren't level-2 compliant never receive multicast datagrams, as they are discarded by the interface hardware itself.

One alternative would be to specify interface name as with the ifreq structure and commands for accessing interface information. ip\_setmoptions and ip\_getmoptions would use ifunit instead of INADDR\_TO\_IFP to locate the interface's ifnet structure.

The high-order 4 bits of a multicast group address are always 1110, so only 5 significant bits are discarded by the mapping function.

The entire ip\_moptions structure must fit in a single mbuf, which limits the size of the structure to 25 bytes. (remember the 20-byte mbuf header). IP\_MAX\_MEMBERSHIPS can be larger but must be less than or equal to 25. ( $4+1+1+2+(4 \times 25) = 104$ )

The datagram is duplicated and two copies are placed in the IP input queue. A multicast application reads from the queue to discard duplicate datagrams.

**12.6**



The process could create a second socket  
**12.8** another IP\_MAX\_MEMBERSHIPS through socket.

Define a new mbuf flag M\_LOCAL for the member of the mbuf header. The flag can loopback packets by ip\_output instead of checksum. ipintr can skip the checksum if flag is on. SunOS 5.X has an option to do (ip\_local\_cksum, page 531, Volume 1).  
**12.9**

There are  $2^{23} - 1$  (8,388,607) unique Ethernet addresses.

**12.10** addresses. Remember that IP group 224 reserved.

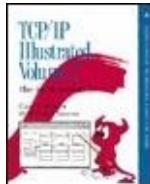
This assumption is correct since `in_addn`  
**12.11** add requests if the interface does not have  
function, and this implies that `in_delmul`  
if `if_ioctl` is null.

The mbuf is never released. It appears that  
`ip_getmoptions` contains a memory leak  
is called from `ip_ctloutput`, which allows

**12.12**

`ip_getmoptions(IP_ADD_MEMBERSHIP)`

which exercises the bug in `ip_getmoption`



[TCP/IP Illustrated, Volume 2: The Implementation](#) By  
Gary R. Wright, W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 13

Responding to an IGMP query from the loopback interface is unnecessary since **13.1** the local host is the only system on the loopback network and it already knows its membership status.

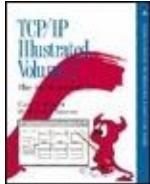
**13.2**  $\text{max\_linkhdr} + \text{sizeof}(\text{struct ip}) + \text{IGMP\_MINLEN} = 16 + 20 + 8 = 44 < 100$

The primary reason for the random

delay in reporting memberships is to minimize (ideally to 1) the number of reports that appear on a multicast network. A point-to-point network consists only of two interfaces, so the delay is not necessary to minimize the response to the query. One interface (presumably a multicast router) generates the query, and the other interface responds.

### 13.3

There is another reason not to flood the interface's output queue with all the membership reports. The output queue may have a packet or byte limit that could be exceeded by many IGMP membership reports. For example, in the SLIP driver, if the output queue is full or the device is too busy, the entire queue of pending packets is discarded ([Figure 5.17](#)).



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 14

**14.1** Five. One each for networks A through E.

**14.2** `grplst_member` is called only by `ip_mforward`, but `ip_mforward` can be called by `ipintr` during protocol processing, or by `ip_output`, which can be called indirectly from the socket layer. The cache is a shared data structure that must be protected while it is being updated.

The membership list itself is protected by splx calls in add\_lgrp and del\_lgrp, where it is modified.

The SIOCDELMULTI command affects only the Ethernet multicast list for the interface. The IP multicast group list remains unchanged, so the interface remains a member of the group. The interface continues accepting multicast datagrams for any groups that are still on the IP group membership list for the interface. Specifically, when ether\_delmulti returns ENETRESET to leioctl, the function lereset is called to reconfigure the interface ([Figure 12.31](#)).

Only one virtual interface is considered to be the parent interface for a multicast spanning tree. If the [packet](#) is accepted on the tunnel, then the physical interface cannot be

the parent and ip\_mforward discards  
the packet.

---

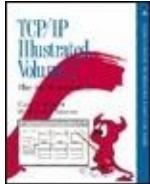
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 15

**15.1** The socket could be shared across a fork or passed to a process through a Unix domain socket ([[Stevens 1990](#)]).

The `sa_len` member of the structure is larger than the size of the buffer after `accept` returns. This is usually not a problem with the fixed-length Internet address, but it can be when using variable-length addresses

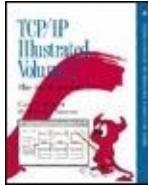
supported by the OSI protocols, for example.

The call to soqremque is only made when so\_qlen is not equal to 0. If **15.4** soqremque returns a null pointer there must be an error in the socket queueing code so the kernel panics.

The copy is made so that bzero can clear the structure while it is locked and so that dom Dispose and sbrelease can be called after splx. **15.5** This minimizes the amount of time the CPU is kept at splimp and therefore the amount of time that network interrupts are blocked.

The sbspace macro will return 0. As a result, the sbappendaddr and sbappendcontrol functions (used by

15.6 UDP) will refuse to queue additional packets. TCP uses sbappend, which assumes that the caller has checked for space first. TCP calls sbappend even when sbspace returns 0. The data placed in the receive queue is not available to a process because the SS\_CANTRCVMORE flag prevents the read system calls from returning any data.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 16

When the value is assigned to uio\_resid in the uio structure it becomes a large negative number. sosend rejects the message with **16.1 EINVAL.**

Net/2 did *not* check for a negative value. This problem is described by the comment at the start of sosend ([Figure 16.23](#)).

No. The only time the cluster is ever filled with less than MCLBYTES is at the end of a message when less than MCLBYTES remain. resid is 0 at this time and the loop is terminated by the break on line 394 before reaching the test for space > 0.

The process blocks until the buffer is unlocked. In this case the lock exists only while another process is examining the buffer or passing data to the protocol layer, and not when a process must wait for space in the buffer, which may take an indefinite amount of time.

If the send buffer contained many mbufs, each of which contained only a few bytes of data, sb\_cc may be well below the limit specified by sb\_hiwater while a large amount of

**16.6** memory would be allocated for the mbufs. If the kernel didn't limit the number of mbufs attached to each buffer, a process could easily create a memory shortage.

recvit is called from recvfrom and recvmsg. Only recvmsg handles control information. The entire msghdr structure, including the length of the control message, is copied back to the process by

**16.7** recvmsg. For address information, recvmsg sets the namelenp argument to null because it expects the length in msg\_nameLEN. When recvfrom calls recvit, the namelenp is nonnull because it expects the length in \*namelenp.

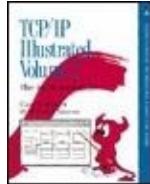
**16.8** MSG\_EOR is cleared by soreceive so that it is not inadvertently returned by soreceive before an M\_EOR mbuf

is processed.

There would be a race condition while select examined the descriptors. If a selectable event

**16.9** occurred after selscan examined the descriptor but before select called tsleep, it would not be detected and the process would sleep until another selectable event occurred.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 17

This simplifies the code that copies data between the kernel and the **17.1** process. copyin and copyout can be used for a single mbuf, but uiomove is needed to handle multiple mbufs.

The code works correctly because **17.2** the first member of a linger structure is the expected integer flag.

---

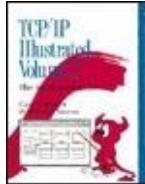
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 18

Write eight rows, one for each possible combination of the bits from the search key, the routing table key, and the routing table mask.

row	1	2	3	1	2	1	&==^	6	&	3
search	table	key	table	3	4?	2	key	mask		
1	0	0	0	0	yes	0	0	=yes		

<b>18.1</b>	2	0	0	1	0 yes	0	0=yes
	3	0	1	0	0 no	1	0=yes
	4	0	1	1	0 no	1	1=no
	5	1	0	0	0 yes	1	0=yes
	6	1	0	1	1 no	1	1=no
	7	1	1	0	0 no	0	0=yes
	8	1	1	1	1 yes	0	0=yes

The column "2 == 4?" should equal the final column "6 & 3." On first glance they are not the same, but we can ignore rows 3 and 7 because in these two rows the routing table bit is 1 while the same bit in the routing table mask is 0. When the routing

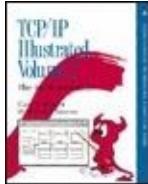
table is built the key is logically ANDed with the mask, guaranteeing that for every bit of 0 in the mask, the corresponding bit in the key is also 0.

Another way to look at the exclusive OR and logical AND in [Figure 18.40](#) is that the exclusive OR becomes 1 only if the search key bit differs from the bit in the routing table key. The logical AND then ignores any differences that correspond to a bit that's 0 in the mask. If the result is still nonzero, the search key does not match the routing table key.

The size of an `rtentry` structure is 120 bytes, which includes the two `radix_node` structures. Each entry [18.2](#) also requires two `sockaddr_in` structures ([Figure 18.28](#)), for 152 bytes per routing table entry. The total is about 3 megabytes.

Since rn\_b is a short integer,  
assuming 16 bits for a short imposes  
**18.3** a limit of 32767 bits per key (4095 bytes).

---



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 19

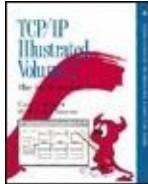
The RTF\_DYNAMIC flag is set in [Figure 19.15](#) when the route is created by a redirect, and the RTF\_MODIFIED flag is set when the **19.1** gateway field of an existing route is modified by a redirect. If a route is created by a redirect and then later modified by another redirect, both flags will be set.

A host route is created for each host

**19.2** accessed through the default route. TCP can then maintain and update routing metrics for each individual host (Figure 27.3).

Each `rt_msghdr` structure requires 76 bytes. Two `sockaddr_in` structures are present for a host route (destination and gateway) giving a message size of 108 bytes. The message size for each ARP entry is 112 bytes: one `sockaddr_in` and one `sockaddr_dl`. The total size is then  $(15 \times 112 + 20 \times 108)$  or 3840 bytes. A network route (instead of a host route) requires an additional 8 bytes for the network mask (116 bytes for the message instead of 108), so if the 20 routes are all network routes, the total size is 4000 bytes.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 20

The return value is returned in the rtm\_errno member of the message ([Figure 20.14](#)) and also as the return value from write ([Figure 20.22](#)). The latter is more reliable since the former may run into mbuf starvation, causing the reply message to be discarded ([Figure 20.17](#)).

For a SOCK\_RAW socket, the pffindproto function ([Figure 7.20](#))

**20.2** returns the entry with a protocol of 0 (the wildcard) if an exact match isn't found.

---

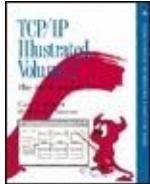
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 21

**21.1** It is assumed that the ifnet structure is at the beginning of the arpcom structure, which it is (Figure 3.20).

Sending the ICMP echo request does not require ARP, since the destination address is the broadcast address. But the ICMP echo replies are normally unicast, so each **21.2** sender uses ARP to determine the

destination Ethernet address. When the local host receives each ARP request, `in_arpinput` replies and creates an entry for the other host.

When a new ARP entry is created, the `rt_gateway` value, a `sockaddr_dl` structure in this case, **21.3** is copied from the entry being cloned by `rtrequest` in [Figure 19.8](#). In [Figure 21.1](#) we see that the `sdl_alen` member of this entry is 0.

With Net/3, if the caller of `arpresolve` supplies a pointer to a routing table entry, `arplookup` is not called, and the corresponding Ethernet address is available through the `rt_gateway` pointer (assuming it hasn't expired). This avoids any type of lookup in the common case. In [Chapter 22](#) we'll **21.4** see that TCP and UDP store a pointer to their routing table entry

in their protocol control block, avoiding a search of the routing table in the case of TCP (where the destination IP address never changes for a connection) and in the case of UDP when the destination doesn't change.

The timeout of an incomplete ARP entry occurs between 0 and 5 minutes after the entry is created. arpresolve sets rt\_expire to the **21.5** current time when the ARP request is sent. The next time arptimer runs, if that entry is not resolved, it is deleted (assuming its reference count is 0).

ether\_output returns EHOSTUNREACH instead of **21.6** EHOSTDOWN, causing an ICMP host unreachable error to be sent to the sending host by ip\_forward.

The value for 140.252.13.32 is set in [Figure 21.28](#) to the current time when the entry is created. It never changes.

The values for 140.252.13.33 and 140.252.13.34 are copied from the entry for 140.252.13.32 when these two entries are cloned by `rrequest`. They are then set to the time at which an ARP request is sent by `arpresolve`, and finally set by `in_arpinput` to the time at which an ARP reply is received, plus 20 minutes.

The value for 140.252.13.35 is also copied from the entry for 140.252.13.32 when the entry is cloned, but then set to 0 by the code at the end of [Figure 21.29](#).

Change the call to `arplookup` at the beginning of [Figure 21.19](#) to always specify a second argument of 1

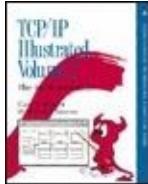
(the create flag).

The first datagram was sent *after* the halfway mark to the next second. Therefore both the first and second datagrams caused ARP

**21.9** requests to be sent, about 500 ms apart, since the kernel's time.tv\_sec variable had different values when these two datagrams were sent.

Each packet to send is an mbuf chain. The m\_nextpkt pointer in the

**21.10** first mbuf in each chain could be used to form a list of mbufs awaiting transmission.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 22

An infinite loop occurs, waiting for a port to become available. This **22.1** assumes the process is allowed to open enough descriptors to tie up all ephemeral ports.

Few, if any, servers support this **22.2** option. [[Cheswick and Bellovin 1994](#)] mention how this would be nice for implementing firewall systems.

The udb structure is initialized to 0 so udb.inp\_lport starts at 0. The first **22.4** time through in\_pcbbind it is incremented to 1, which is less than 1024, so it is set to 1024.

Normally the caller sets the address family (sa\_family) to AF\_INET, but we saw in [Figure 22.20](#) that the test for this is commented out. The caller can set the length member (sa\_len), but we saw in [Figure 15.20](#) that the function sockargs always sets this to the third argument to bind, which for a sockaddr\_in structure is specified as 16, normally using C's sizeof operator.

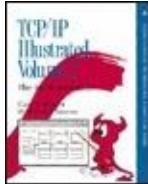
**22.5** The local IP address (sin\_addr) can be specified as a wildcard address or as a local IP address. The local port number (sin\_port), can be either 0 (telling the kernel to choose an ephemeral port) or nonzero if the

process wants a particular port. Normally a TCP or UDP server specifies a wildcard IP address and a nonzero port, and a UDP client often specifies a wildcard IP address and a port number of 0.

A process is allowed to bind a local broadcast address, because the call to `ifa_ifwithaddr` in [Figure 22.22](#) succeeds. That address is used as the source address for IP datagrams sent on the socket. As noted in [Section C.2](#), this behavior is not allowed by RFC 1122.

An attempt to bind 255.255.255.255, however, fails, since that address is not acceptable to `ifa_ifwithaddr`.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 23

sosend places the user data into a single mbuf if the size is less than or equal to 100 bytes; into two mbufs if the size is less than or equal to 207 bytes; or into one or more mbufs, each with a cluster, otherwise. Furthermore, sosend calls MH\_ALIGN if the size is less than 100 bytes, which, it is hoped, will allow room at the beginning of the mbuf for the protocol headers. Since udp\_output calls M\_PREPEND, the following five

scenarios are possible: (1) If the size of the user data is less than or equal to 72 bytes, a single mbuf contains the IP header, UDP header, and data. (2) If the size is between 73 and 100 bytes, one mbuf is

**23.1** allocated by sosend for the data and another is allocated by M\_PREPEND for the IP and UDP headers. (3) If the size is between 101 and 207 bytes, two mbufs are allocated by sosend for the data and another by M\_PREPEND for the IP and UDP headers. (4) If the size is between 208 and MCLBYTES, one mbuf with a cluster is allocated by sosend for the data and another by M\_PREPEND for the IP and UDP headers. (5) Beyond this size, sosend allocates as many mbufs with clusters as necessary to hold the data (up to 64 for a maximum data size of 65507 bytes with 1024-byte clusters), and one mbuf is allocated by M\_PREPEND for the IP and UDP headers.

IP options are passed to `ip_output`, which calls `ip_insertoptions` to insert the options into the outgoing IP datagram. This function in turn allocates a new mbuf to hold the IP header including options if the first mbuf in the chain points to a cluster (which never happens with UDP output) or if there is not enough room at the beginning of the first mbuf in the chain for the options. In scenario 1 from the previous solution, the size of the options determines whether another mbuf is allocated by `ip_insertoptions`: if the size of the user data is less than 10028 *optlen*, **23.2** (where *optlen* is the number of bytes of IP options), there is room in the mbuf for the IP header with options, the UDP header, and the data.

In scenarios 2, 3, 4, and 5, the first mbuf in the chain is always allocated by `M_PREPEND` just for the IP and UDP headers. `M_PREPEND` calls `m_prepend`,

which calls MH\_ALIGN, moving the 28 bytes of headers to the end of the mbuf, hence there is always room for the maximum of 40 bytes of IP options in this first mbuf in the chain.

No. The function in\_pcbconnect is called, either when the application calls connect or when the first datagram is sent on an unconnected UDP socket. Since the **23.3** local address is a wildcard and the local port is 0, in\_pcbconnect sets the local port to an ephemeral port (by calling in\_pcbbind) and sets the local address based on the route to the destination.

The processor priority level is left at **23.4** splnet; it is not restored to the saved value. This is a bug.

No. `in_pcbconnect` will not allow a connection to port 0. Even if the process doesn't call `connect` **23.5**, an implicit connect is performed, so `in_pcbconnect` is called regardless.

The application must call `ioctl` with the `SIOCGIFCONF` command to return information on all configured IP interfaces. The destination address in the received UDP datagram must then be compared **23.6** against all the IP addresses and broadcast addresses in the list returned by `ioctl`. (As an alternative to `ioctl`, the `sysctl` system call described in [Section 19.14](#) can also be used to obtain the information on all the configured interfaces.)

`recvit` releases the mbuf with the **23.7** control information.

To disconnect a connected UDP socket, call connect with an invalid address, such as 0.0.0.0, and a port of 0. Since the socket is already connected, soconnect calls sodisconnect, which calls udp\_usrreq with a PRU\_DISCONNECT request. This sets the foreign address to 0.0.0.0 and the foreign port to 0, allowing a subsequent call to sendto that specifies a destination address to succeed. Specifying the invalid address causes the PRU\_CONNECT request from sodisconnect to fail. We don't want the connect to succeed, we just want the PRU\_DISCONNECT request executed and this back door through connect is the only way to execute this request, since the sockets API doesn't provide a disconnect function.

**23.8**

The manual page for connect(2)

usually contains the following note that hints at this: "Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address." What this note fails to mention is that the call to connect for the invalid address is expected to return an error. The term *null address* is also vague: it means the IP address 0.0.0.0, not a null pointer for the second argument to bind.

Since an unconnected UDP socket is temporarily connected to the foreign IP address by `in_pcbconnect`, the scenario is the same as if the process calls **23.9** `connect`: the datagram is sent out the primary interface with a destination IP address corresponding to the broadcast address of that interface.

The server must set the IP\_RECVSTADDR socket option and use recvmsg to obtain the destination IP address from the client's request. For this address to **23.10** be the source IP address of the reply requires that this IP address be bound to the socket. Since you cannot bind a socket more than once, the server must create a brand new socket for each reply.

Notice in ip\_output ([Figure 8.22](#)) that IP does not modify the DF bit supplied by the caller. A new socket option could be defined to cause udp\_output to set the DF bit before passing datagrams to IP. **23.11**

No. It is used only in the udp\_input **23.12** function and should be local to that function.

---

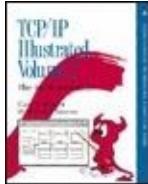
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 24

**24.1** The total number of ESTABLISHED connections is 126,820. Dividing this into the total number of bytes transmitted and received yields an average of about 30,000 bytes in each direction.

In `tcp_output`, the mbuf obtained for the IP and TCP headers also contains room for the link-layer headers (`max_linkhdr`). The IP and TCP

header prototype is copied into the mbuf using bcopy, which won't work

**24.2** if the 40-byte header were split between two mbufs. Although the 40-byte headers must fit into one mbuf, the link-layer header need not. But a performance penalty would occur later (`ether_output`) because a separate mbuf would be required for the link-layer header.

On the author's system `bsdi`, the count was 16, 15 of which were standard system daemons (Telnet, Rlogin, FTP, etc.). On

`vangogh.cs.berkeley.edu`, a medium-sized multiuser

**24.3** system with around 20 users, the count was 60. On a large multiuser system (`world.std.com`) with around 150 users, the count was 417 TCP end points and 809 UDP end points.

---

**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)

---

## Appendix A. Solutions to Selected Exercises

---

# Chapter 25

In Figure 24.5 there were 531,285 delayed ACKs over 30 days, or one every 5 seconds, or one delayed ACK every 150 ms. This means 96% of the time (24 ticks) the TCP control block is checked for the delay timer to be set. On the large multiuser system in the figure this involves looking at over 400 control blocks.

One alternative implementation would be to have the delayed ACK is needed and only go through the loop when the flag is set. Alternatively, another approach is to have a linked list that contains only the control blocks that need to be checked. For example, the variable igmp\_timers\_array contains a linked list of pointers to the TCP control blocks.

This allows the variable `tcp_keepintvl` to be set by the kernel, which then changes the value of `tcp_slowtimo` when `tcp_idletimer` is called.

**25.3** `t_idle` actually counts the time since a segment was transmitted. This is because TCP output needs to know when the other end has received the ACK or the receipt of a data segment ([Figure 28.8](#)).

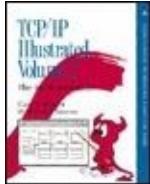
Here is one way to rewrite the code:

```
case TCPT_2MSL:  
    if (tp->t_state == TCPS_SYN_SENT)  
        tp = tcp_close(tp);  
    else {  
        if (tp->t_idle <= tcp_keepintvl)  
            tp->t_timer[TCPT_SYN] = 0;  
        else  
            tp = tcp_close(tp);  
    }  
break;
```

When the duplicate ACK is received, `t_idle` is set to 1198 ticks. When the `FIN_WAIT_2` timer expires, `t_idle` will be 1198 ticks so the timer is set to 150 ticks. When the `TIME_WAIT` timer expires the next time, `t_idle` will be 1198 ticks so the connection is closed. The duplicate ACK example shows how the connection is closed.

The first keepalive probe will be sent 1 hour after the connection is established. If the process sets the option, nothing happens. The `SO_KEEPALIVE` option in the socket structure is set to 1 hour in the future, since the option is effective after 25.16 seconds. The `tcp_sendkeepalive` function sends the first probe.

The value of `tcp_rttdflt` initializes the RTT for a new connection. A site can change the default value by changing the global variable. If the value were a #define, it would have to be changed only by recompiling the kernel.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 26

The counter `t_idle` is always running for a connection, whereas

**26.1** TCP does not measure the amount of time since the last segment was sent on a connection.

In Figure 25.26 `snd_nxt` is set to

**26.2** `snd_una`, giving a value of 0 for `len`.

If you're running a Net/3 system and encounter a peer that can't handle either of these two newer options (i.e., that peer refuses to establish the connection, even though a host is required to ignore options it doesn't understand), this global can be patched in the kernel to disable one or both of these options.

The timestamp option would have updated the RTT estimators each time an ACK was received for new data: 16 times, twice the number of times without the option. The value calculated when the ACK of 6145 was received at time 217.944, however, would have been bogus either the data segment with bytes 5633 through 6144 that was sent at time 3.740, or the received ACK of 6145, was delayed somewhere for about 200 seconds.

There is no guarantee that the 2-byte MSS value is correctly aligned for such a memory reference.

(This solution is from Dave Borman.) The maximum amount of TCP data in a segment is 65495 bytes, which is 65535 minus the minimum IP and TCP headers (40). Hence there are 39 values of the urgent offset that make no sense: 65496 through and including 65535. Whenever the sender has a 32-bit urgent offset that exceeds 65495, 65535 is sent as the urgent offset instead, and the URG flag is set. This puts the receiver into urgent mode and tells the receiver that the urgent offset points to data that has not been sent yet. The special value of 65535 continues to be sent as the urgent offset (with the URG flag set) until the urgent offset is less than or equal to 65495, at which point the real

urgent offset is sent.

## 26.7

We've mentioned that data segments are transmitted reliably (i.e., the retransmission timer is set) but ACKs are not. RST segments are not transmitted reliably either. RST segments are generated when a bogus segment arrives (either a segment that is wrong for a connection, or a segment for a nonexistent connection). If the RST segment is discarded by ip\_output, when the other end retransmits the segment that caused the RST to be generated, another RST will be generated.

The application does eight writes of 1024 bytes. The first four times sosend is called, tcp\_output is called, and a segment is sent. Since these four segments each contain

the final bytes of data in the send buffer, the PSH flag is set for each segment ([Figure 26.25](#)). The send buffer is also full, so the next write by the process puts the process to sleep in `sosend`. When the ACK is returned with an advertised window **26.8** of 0, the 4096 bytes of data in the send buffer have been acknowledged and are discarded, and the process wakes up and continues filling the send buffer with the next four writes. But nothing can be sent until a nonzero window is advertised by the receiver. When this happens, the next four segments are sent, but only the final segment contains the PSH flag, since the first three segments do not empty the send buffer.

The `tp` argument to `tcp_respond` can be a null pointer if the segment being sent does not correspond to a connection. The code should check **26.9**

the value of tp and use the default only if the pointer is null.

tcp\_output always allocates an mbuf just to contain the IP and TCP headers, by calling MGETHDR in [Figures 26.25 and 26.26](#). This code allocates room at the front of the new mbuf only for the link-layer header (max\_linkhdr). If IP options are in use and the size of the options exceeds max\_linkhdr, another mbuf is allocated by ip\_insertoptions. If the size of the

**26.10** IP options is less than or equal to max\_linkhdr, then even though ip\_insertoptions will use the space at the beginning of the mbuf, this will cause ether\_output to allocate another mbuf for the link-layer header (assuming Ethernet output).

To try to avoid the extra mbuf, [Figures 26.25 and 26.26](#) could call MH\_ALIGN if the segment will contain IP options.

About 80 lines of C code, assuming RFC 1323 timestamps are in use and the segment is timed.

## 26.11

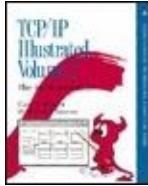
The macro MGETHDR invokes the macro MALLOC, which might call the function malloc. The function m\_copy is also called, but a full-sized segment will be in a cluster, so the mbuf is not copied, a reference is made to the cluster. The call to MGET by m\_copy might call malloc. The function bcopy copies the header template and in\_cksum calculates the TCP checksum.

Nothing changes with writev because of the logic in sosend. Since the total size of the data (150) is less than MINCLSIZE (208), one mbuf is allocated for the first 100 bytes, and since the protocol is not atomic, the

PRU\_SEND request is issued. Another mbuf is allocated for the next 50 bytes, and another PRU\_SEND is issued. TCP still **26.12** generates two segments. (writev only generates a single "record," that is, a single PRU\_SEND request, for PR\_ATOMIC protocols such as UDP.)

With two buffers of length 200 and 300 the total size now exceeds MINCLSIZE. An mbuf cluster is allocated and only one PRU\_SEND is issued. One 500-byte segment is generated by TCP.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 27

The first six rows of the table are asynchronous errors that are generated by the receipt of a segment or the expiration of a timer. By storing the nonzero error code in `so_error`, the process receives the error on the next read or write. The call from `tcp_disconnect`, however, occurs when the process calls `close`, **27.1** or when the descriptor is closed automatically on process termination. In either case of the descriptor being closed, the process

won't issue a read or write call to fetch the error. Also, since the process had to set the socket option explicitly to force the RST, returning an error provides no useful information to the process.

Assuming a 32-bit u\_long, the **27.2** maximum value is just under 4298 seconds (1.2 hours).

The statistics in the routing table are updated by `tcp_close` and it is called only when the connection enters the CLOSED state. Since the sending of data to the other end is terminated by the FTP client (it does the active close), the local end point enters the TIME\_WAIT state. The routing table statistics won't be updated until twice the MSL has elapsed.

---

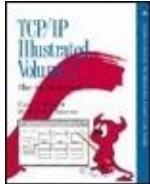
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 28

**28.1** 0, 1, 2, and 3.

34.9 Mbits/sec. For higher speeds,  
**28.2** larger buffers are required on both ends.

In the general case, `tcp_dooptions` doesn't know whether the two timestamp values are aligned on 32-bit boundaries or not. The

**28.3** special code in [Figure 28.4](#), however, knows that the values are on 32-bit boundaries, and avoids calling bcopy.

The "options prediction" code in [Figure 28.4](#) handles only the recommended format, so systems that send other than the recommended format cause the slower processing of `tcp_dooptions` to occur for every received segment.

**28.5** If `tcp_template` were called every time a socket were created, instead of every time a connection is established, each listening server on a system would have one allocated, which it would never use.

The timestamp clock frequency

should be between 1 bit/ms and 1 bit/sec. (Net/3 uses 2 bits/sec.)

**28.6** With the highest frequency of 1 bit/ms, a 32-bit timestamp wraps its sign bit in  $2^{31} / (24 \times 60 \times 60 \times 1000)$  days, which is 24.8 days.

With a frequency of 1 bit per 500 ms, a 32-bit timestamp wraps its sign bit in  $2^{31} / (24 \times 60 \times 60 \times 2)$

**28.7** days, which is 12,427 days, or about 34 years, longer than the uptime of current computer systems.

The cleanup function of an RST should take precedence over timestamps, and it is recommended

**28.8** that RSTs not carry timestamps (which is enforced by `tcp_input` in Figure 26.24).

Since the client is in the ESTABLISHED state, processing ends up in [Figure 28.24](#). `todrop` is 1 because `rcv_nxt` was incremented over the SYN when it was first received. The SYN flag is cleared (since it is a duplicate), `ti_seq` is incremented, and `todrop` is decremented to 0. The if statement at the top of [Figure 28.25](#) is executed since `todrop` and `ti_len` are both 0. The next if statement is skipped, and processing continues with the call to `m_adj`. But `tcp_output` is not called in the continuation of `tcp_input` in the next chapter, therefore the client does not respond to the duplicate SYN/ACK. The server will time out and resend the SYN/ACK (recall the timer set in [Figure 28.17](#) when a passive socket receives a SYN), which will also be ignored. This is another bug in the code in [Figure 28.25](#) and this one is also fixed with the code shown in [Figure 28.30](#).

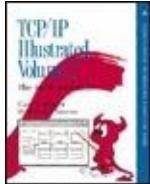
## 28.9

## **28.10**

The client's SYN arrives at the server and is delivered to the socket in the TIME\_WAIT state. The code in [Figure 28.24](#) turns off the SYN flag and the code in [Figure 28.25](#) jumps to dropafterack, dropping the segment but generating an ACK with an acknowledgment field of `rcv_nxt` ([Figure 26.27](#)). This is called a *resynchronization ACK* because its purpose is to tell the other end what sequence number it expects. When this ACK is received at the client (which is in the SYN\_SENT state), its acknowledgment field is not the expected value ([Figure 28.18](#)), causing an RST to be sent. The sequence number of the RST is the acknowledgment field from the resynchronization ACK, and the ACK flag of the RST segment is off ([Figure 29.28](#)). When the server receives the RST, its TIME\_WAIT state is prematurely terminated and the socket is closed on the server's host ([Figure 28.36](#)). The client

times out after 6 seconds and retransmits its SYN. Assuming a listening server process is running on the server host, the new connection is established. Because of this form of TIME\_WAIT assassination, a new connection is established not only when a SYN arrives with a higher sequence number (as checked for in [Figure 28.29](#)), but also when a SYN with a lower sequence number arrives.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 29

Assume a 2-second RTT. The server has a passive open pending and the client issues its active open at time 0. The server receives the SYN at time 1 and responds with its own SYN and an ACK of the client's SYN. The client receives this segment at time 2, and the code in [Figure 28.20](#) completes the active open with the **29.1** call to soisconnected (waking up the client process) and an ACK will be sent back to the server. The server receives the ACK at time 3, and the

code in [Figure 29.2](#) completes the server's passive open, returning control to the server process. In general, the client process receives control about one-half RTT before the server.

Assume the sequence number of the SYN is 1000 and the 50 bytes of data are numbered 10011050. When the SYN is processed by `tcp_input`, first the case starting in [Figure 28.15](#) is executed, which sets `rcv_nxt` to 1001, and then a jump is made to step6. [Figure 29.22](#) calls `tcp_reass` and the data is placed onto the socket's reassembly queue. But the data cannot be appended to the **29.2** socket's receive buffer yet ([Figure 27.23](#)) so `rcv_nxt` is left at 1001. When `tcp_output` is called to generate the immediate ACK, `rcv_nxt` (1001) is sent as the acknowledgment field. In summary, the SYN is acknowledged, but not the 50 bytes of data. Since the client

will retransmit the 50 bytes of data, there is no advantage in sending data with a SYN generated by an active open.

The server's socket is in the SYN\_RCVD state when the client's ACK/FIN arrives, so `tcp_input` ends up processing the ACK in [Figure 29.2](#). The connection moves to the ESTABLISHED state and `tcp_reass` appends the already-queued data to the socket's receive buffer. `recv_nxt` is incremented to 1051. `tcp_input` continues and the FIN is handled in [Figure 29.24](#) where the `TF_ACKNOW` flag is set and `recv_nxt` becomes 1052. `socantrcvmore` sets the socket's state so that after the server reads the 50 bytes of data, the server will receive an end-of-file. The server's socket also moves to the CLOSE\_WAIT state. `tcp_output` will be called to ACK the client's FIN (since `recv_nxt` equals 1052).

**29.3**

Assuming the server process closes its socket when it reads the end-of-file, the server will then send a FIN for the client to ACK.

In this example six segments requiring three round trips are required to pass the 50 bytes from the client to server. To reduce the number of segments requires the TCP extensions for transactions [Braden 1994].

The client's socket is in the SYN\_SENT state when the server's response is received. Figure 28.20 processes the segment and moves the connection to the ESTABLISHED state. A jump is made to step6 and the data is processed in Figure 29.22. TCP\_REASS appends the data to the socket's receive buffer and rcv\_nxt is incremented to acknowledge the data. The FIN is then processed in Figure 29.24, incrementing rcv\_nxt again and 29.4

moving the connection to the CLOSE\_WAIT state. When `tcp_output` is called, the acknowledgment field ACKs the SYN, the 50 bytes of data, and the FIN. The client process then reads the 50 bytes of data, followed by the end-of-file, and then probably closes its socket. This moves the connection to the LAST\_ACK state and causes a FIN to be sent by the client, which the server should acknowledge.

The bug is in the entry `tcp_outflags[TCPS_CLOSING]` shown in [Figure 24.16](#). It specifies the TH\_FIN flag, whereas the state transition diagram ([Figure 24.15](#)) doesn't specify that the FIN should

**29.5** be retransmitted. To fix this, remove TH\_FIN from the `tcp_outflags` entry for this state. The bug is relatively harmlessit just causes two extra segments to be exchangedand a simultaneous close or a close following a self-connect is rare.

No. An OK return from a write system call only means the data has been copied into the socket buffer.

**29.6** Net/3 does not notify the process when that data is acknowledged by the other end. An application-level acknowledgment is required to obtain this information.

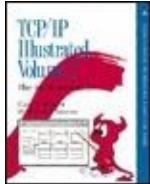
**29.7** RFC 1323 timestamps defeat header compression because whenever the timestamps change, the TCP options change, and the segment is sent uncompressed. The window scale option has no effect because the value in the TCP header is still a 16-bit value.

IP assigns the ID field from a global variable that is incremented each time *any* IP datagram is sent. This

increases the probability that two consecutive TCP segments sent on **29.8** the same connection will have ID values that differ by more than 1. A difference other than 1 causes the *Dipid* field in [Figure 29.34](#) to be transmitted, increasing the size of the compressed header. A better scheme would be for TCP to maintain its own counter for assigning IDs.

---





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 30

**30.2**

Yes, the RST is still sent. Part of process termination is the closing of all open descriptors. The same function (soclose) is eventually called, regardless of whether the process explicitly closes the socket descriptor or implicitly closes it (by terminating first).

No. The only use of this constant is

when a listening socket sets the SO\_LINGER socket option with a linger time of 0. Normally this causes

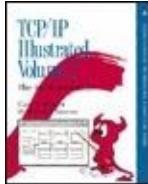
**30.3** an RST to be sent when the connection is closed ([Figure 30.12](#)), but [Figure 30.2](#) changes this value of 0 to 120 (clock ticks) for a listening socket that receives a connection request.

Two if this is the first use of the default route; otherwise one. When the socket is created the Internet PCB is set to 0 by in\_pcalloc. This sets the route structure in the PCB to 0. When the first segment is sent (the SYN), tcp\_output calls ip\_output. Since the ro\_rt pointer is null, ro\_dst is filled in with the destination address of the IP datagram and rtalloc is called. The

**30.4** pointer to the default route is saved in the ro\_rt member of the route structure within the PCB for this connection. When ether\_output is called by ip\_output, it checks

whether the `rt_gwroute` member of the routing table entry is null, and, if so, `rtalloc1` is called. Assuming the route doesn't change, each time `tcp_output` is called for this connection, the cached `ro_rt` pointer is used, avoiding any additional routing table lookups.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

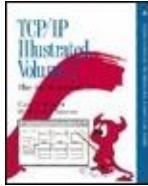
# Chapter 31

**31.1** Because catchpacket will always run to completion before any sleeping processes are awakened by the bpf\_wakeup call.

**31.2** A process that opens a BPF device may call fork resulting in multiple processes with access to the same BPF device.

**31.3** Only supported devices are on the BPF interface list (`bpf_iflist`), so `bpf_setif` returns ENXIO when the interface is not found.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix A. Solutions to Selected Exercises

# Chapter 32

0 in the first example, and 255 in the second. Both of these values are reserved in RFC 1700 [[Reynolds and Postel 1994](#)] and should not appear in datagrams. This means, for **32.1** example, that a socket created with a protocol of IPPROTO\_RAW should always have the IP\_HDRINCL socket option set, and datagrams written to the socket should have a valid protocol value.

Since the IP protocol value of 255 is reserved, datagrams should never appear on the wire with this protocol value. Since this is a nonzero protocol value, the first of the three tests in rip\_input will ignore every received datagram that does not have this protocol value. Therefore the process should not receive any datagrams on the socket.

**32.2** Even though this protocol value is reserved and datagrams should never appear on the wire with this value, the first of the three tests in rip\_input allows datagrams with any protocol value to be received by sockets of this type. The only input filtering that occurs for this type of raw socket is based on the source and destination IP addresses, if the process calls either connect or bind, or both.

Since the array ip\_protox array ([Figure 7.22](#)) contains information about which protocol the kernel supports, the ICMP error should be generated only when there are no raw listeners for the protocol and the pointer inetsw[ip\_protox[ip->ip\_p]].pr\_input equals rip\_input.

In both cases the process must build its own IP header, in addition to whatever follows the IP header (UDP datagram, TCP segment, or whatever). With a raw IP socket, output is normally done using sendto specifying the destination address as an Internet socket address structure containing an IP address. ip\_output is called and normal IP routing is done based on the destination IP address.

BPF requires the process to supply a complete data-link header, such as an Ethernet header. Output is normally done by calling write, since

a destination address cannot be specified. The packet is passed directly to the interface output function, bypassing `ip_output` ([Figure 31.20](#)). The process selects the outgoing interface using the `BIOCSETIF` ioctl ([Figure 31.16](#)). Since IP routing is not performed, the destination of the packet is limited to another system on an attached network (unless the process duplicates the IP routing function and sends the packet to a router on an attached network, for the router to forward based on the destination IP address).

A raw IP socket receives only IP datagrams destined for an IP protocol that the kernel does not process itself. A process cannot receive TCP segments or UDP datagrams on a raw socket, for example.

## 32.6

BPF can receive *all* frames received

on a specified interface, regardless of whether they are IP datagrams or not. The BIOCPROMISC ioctl can put the interface into a promiscuous mode, to receive datagrams that are not even destined for this host.

---

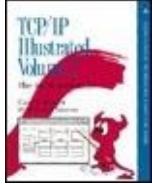
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Appendix B. Source Code Availability

URLs: Uniform Resource Locators

4.4BSD-Lite

Operating Systems that Run the  
4.4BSD-Lite Networking Software

RFCs

GNU Software

PPP Software

mouted Software

ISODE Software



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Ga  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

# URLs: Uniform Resource Locators

This text uses URLs to specify the location and resources on the Internet. For example, the colon technique is designated as

`ftp://ftp.cdrom.com/pub/bsd-sources/`

This specifies anonymous FTP to the host `ftp`. filename is `4.4BSDLite.tar.gz` in the `pub/bsdsources`. The suffix `.tar` implies Unix `tar(1)` format, and the addition `.gz` implies that the file has been compressed by `gzip(1)` program.

---

[TCP/IP Illustrated, Volume 2: The Implementation](#) By Ga  
W. Richard Stevens  
[Table of Contents](#)

## **Appendix B. Source Code Availability**

---

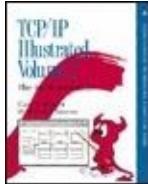
### **4.4BSD-Lite**

There are numerous ways to obtain the 4.4BSD  
entire 4.4BSD-Lite release is available from Wa

<ftp://ftp.cdrom.com/pub/bsd-sources/>

You can also obtain this release on CD-ROM. Co  
or +1 510 674 0783.

O'Reilly & Associates publishes the entire set of  
with the 4.4BSD-Lite release on CD-ROM. Cont  
+1 707 829 0515.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

# Operating Systems that Run the 4.4BSD-Lite Networking Software

The 4.4BSD-Lite release is *not* a complete operating system. To experiment with the networking software described in this text you need an operating system that is built from the 4.4BSD-Lite release or an environment that supports the 4.4BSD-Lite networking code.

The operating system used by the authors is commercially available from Berkeley Software Design, Inc. Contact 1 800 ITS BSD8, +1 719 260 8114, or

[info@bsdi.com](mailto:info@bsdi.com) for additional information.

There are also freely available operating systems built on 4.4BSD-Lite. These are known by the names NetBSD, 386BSD, and FreeBSD. Additional information is available from Walnut Creek CD-ROM ([ftp.cdrom.com](ftp://ftp.cdrom.com)) or on the various comp.os.386bsd Usenet newsgroups.

---

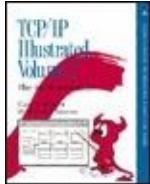
Team-Fly



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

## RFCs

All RFCs are available at no charge through electronic mail or by using anonymous FTP across the Internet. Sending electronic mail as shown here:

To: rfc-info@ISI.EDU  
Subject: getting rfcs

help: ways\_to\_get\_rfcs

returns a detailed listing of various ways to obtain the RFCs using either email or anonymous FTP.

Remember that the starting place is to

obtain the current index and look up the RFC that you want in the index. This entry tells you if that RFC has been made obsolete or updated by a newer RFC.

---

**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)

[TCP/IP Illustrated, Volume 2: The Implementation](#) By Ga  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

### GNU Software

The GNU Indent program was used to format a presented in the text, and the GNU Gzip program on the Internet to compress files. These programs as

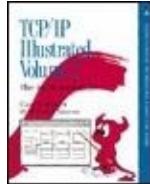
<ftp://prep.ai.mit.edu/pub-gnu/indent>  
<ftp://prep.ai.mit.edu/pub-gnu/gzip-1>

The numbers in the filenames will change as new released. There are also versions of the Gzip program for operating systems, such as MS-DOS.

There are many sites around the world that also have GNU archives, and the FTP greeting on [prep.ai](http://prep.ai) lists their names.

**Team-Fly**





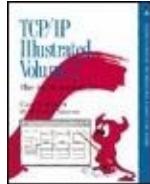
[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

## PPP Software

There are several freely available implementations of PPP. Part 5 of the comp.protocols.ppp FAQ is a good place to start:

<http://cs.uni-bonn.de/ppp/part5.html>



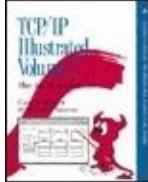
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

## mROUTED Software

Current releases of the mROUTED software as well as other multicast applications can be found at the Xerox Palo Alto Research Center:

<ftp://parcftp.xerox.com/pub/net-research/>



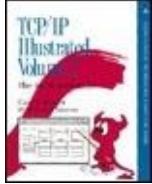
[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix B. Source Code Availability

## ISODE Software

An SNMP agent implementation compatible with Net/3 is part of the ISODE software package. For more information, start with the ISODE Consortium's World Wide Web page at

<http://www.isode.com/>



**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

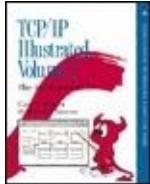
# Appendix C. RFC 1122 Compliance

This appendix summarizes the compliance of the Net/3 implementation with RFC 1122 [Braden 1989a]. This RFC summarizes these requirements in four categories

- link layer
- internet layer
- UDP
- TCP

We have chosen to present these requirements in the same breakdown and order as the chapters of this text.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.1 Link-Layer Requirements

This section summarizes the link-layer requirements from Section 2.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *May support trailer encapsulation.*

Partially: Net/3 does not send IP datagrams with trailer encapsulation but some Net/3 device drivers may be able to receive such datagrams. We have omitted all the trailer encapsulation code in this text. Interested readers are referred to RFC 893 and Section 11.8 of [Leffler et al.

[1989](#)] for additional details.

- *Must* not send trailers by default without negotiation.

Not applicable: Net/2 would negotiate the use of trailers but Net/3 ignores requests to send trailers and does not request trailers itself.

- *Must* be able to send and receive RFC 894 Ethernet encapsulation.

Yes: Net/3 supports RFC 894 Ethernet encapsulation.

- *Should* be able to receive RFC 1042 (IEEE 802) encapsulation.

No: Net/3 processes packets received with 802.3 encapsulation but only for use with OSI protocols. IP packets that arrive with 802.3 encapsulation are discarded by ether\_input ([Figure 4.13](#)).

- *May* send RFC 1042 encapsulation, in which case there must be a software configuration switch to select the encapsulation method and RFC 894

must be the default.

No: Net/3 does not send IP packets in RFC 1042 encapsulation.

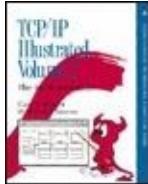
- *Must* report link-layer broadcasts to the IP layer.

Yes: The link layer reports link-layer broadcasts by setting the M\_BCAST flag (or the M\_MCAST flag for multicasts) in the mbuf packet header.

- *Must* pass the IP TOS value to the link layer.

Yes: The TOS value is not passed explicitly, but is part of the IP header available to the link layer.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.2 IP Requirements

This section summarizes the IP requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must implement IP and ICMP.*

Yes: `inetsw[0]` implements the IP protocol and `inetsw[4]` implements ICMP.

- *Must handle remote multihoming in application layer.*

Yes: The kernel is unaware of communication to remote multihomed

hosts and neither hinders nor supports such communication by an application.

- *May support local multihoming.*

Yes: Net/3 supports multiple IP interfaces with the ifnet list and multiple addresses per IP interface with the ifaddr list for each ifnet structure.

- *Must meet router specifications if forwarding datagrams.*

Partially: See [Chapter 18](#) for a discussion of the router requirements.

- *Must provide configuration switch for embedded router functionality. The switch must default to host operation.*

Yes: The ipforwarding variable defaults to false and controls the IP packet forwarding mechanism in Net/3.

- *Must not enable routing based on number of interfaces.*

Yes: The if\_attach function does not modify ipforwarding according to the

number of interfaces configured at system initialization time.

- *Should* log discarded datagrams, including the contents of the datagram, and record the event in a statistics counter.

Partially: Net/3 does not provide a mechanism for logging the contents of discarded datagrams but maintains a variety of statistics counters.

- *Must* silently discard datagrams that arrive with an IP version other than 4.

Yes: ipintr implements this requirement.

- *Must* verify IP checksum and silently discard an invalid datagram.

Yes: ipintr calls in\_cksum and implements this requirement.

- *Must* support subnet addressing (RFC 950).

Yes: Every IP address has an associated

subnet mask in the `in_ifaddr` structure.

- *Must* transmit packets with host's own IP address as the source address.

Partially: When the transport layer sends an IP datagram with all-0 bits as the source address, IP inserts the IP address of the outgoing interface in its place. A process can bind one of the local IP broadcast addresses to the local socket, and IP will transmit it as an invalid source address.

- *Must* silently discard datagrams not destined for the host.

Yes: If the system is not configured as a router, `ipintr` discards datagrams that arrive with a bad destination address (i.e., an unrecognized unicast, broadcast, or multicast address).

- *Must* silently discard datagrams with bad source address (nonunicast address).

No: `ipintr` does not examine the source address of incoming datagrams before

delivering the datagram to the transport protocols.

- *Must* support reassembly.

Yes: ip\_reass implements reassembly.

- *May* retain same ID field in identical datagrams.

No: ip\_output assigns a new ID to every outgoing datagram and does not allow the ID to be specified by the transport protocols. See [Chapter 32](#).

- *Must* allow the transport layer to set TOS.

Yes: ip\_output accepts any TOS value set in the IP header by the transport protocols. The transport layer must default TOS to all 0s. The TOS value for a particular datagram or connection may be set by the application through the IP\_TOS socket option.

- *Must* pass received TOS up to transport layer.

Yes: Net/3 preserves the TOS field during input processing. The entire IP header is made available to the transport layer when IP calls the `pr_input` function for the receiving protocol. Unfortunately, the UDP and TCP transport layers ignore it.

- *Should not* use RFC 795 [[Postel 1981d](#)] link-layer mappings for TOS.

Yes: Net/3 does not use these mappings.

- *Must not* send packet with TTL of 0.

Partially: The IP layer (`ip_output`) in Net/3 does not check this requirement and relies on the transport layers not to construct an IP header with a TTL of 0. UDP, TCP, ICMP, and IGMP all select a nonzero TTL default value. The default value can be overridden by the `IP_TTL` option.

- *Must not* discard received packets with a TTL less than 2.

Yes: If the system is the final

destination of the packet, ipintr accepts it regardless of the TTL value. The TTL is examined only when the packet is being forwarded.

- *Must* allow transport layer to set TTL.

Yes: The transport layer must set TTL before calling ip\_output.

- *Must* enable configuration of a fixed TTL.

Yes: The default TTL is specified by the global integer ip\_defttl, which defaults to 64 (IPDEFTTL). Both UDP and TCP use this value unless the IP\_TTL socket option has specified a different value for a particular socket. ip\_defttl can be modified through the IPCTL\_DEFSTTL name for sysctl.

## Multihoming

- *Should* select, as the source address for a reply, the specific address received as the destination address of the request.

Yes: Responses generated by the kernel (ICMP reply messages) include the correct source address ([Section C.5](#)). Responses generated by the transport protocols are described in their respective chapters.

- *Must* allow application to choose local IP address.

Yes: An application can bind a socket to a specific local IP address ([Section 15.8](#)).

- *May* silently discard datagrams addressed to an interface other than the one on which it is received.

No: Net/3 implements the weak end system model and ipintr accepts such packets.

- *May* require packets to exit the system through the interface with an IP address that corresponds to the source address of the packet. This requirement pertains only to packets that are not source routed.

No: Net/3 allows packets to exit the system through any interface another weak end system characteristic.

## Broadcast

- *Must* not select an IP broadcast address as a source address.

Partially: If an application explicitly selects a source address, the IP layer does not override the selection.

Otherwise, IP selects as a source address the specific IP address associated with the outgoing interface.

- *Should* accept an all-0s or all-1s broadcast address.

Yes: ipintr accepts packets sent to either address.

- *May* support a configurable option to send all 0s or all 1s as the broadcast address on an interface. If provided, the configurable broadcast address *must* default to all 1s.

No: A process must explicitly send to either the all-0s (INADDR\_ANY) or all-1s broadcast address (INADDR\_BROADCAST). There is no configurable default.

- *Must* recognize all broadcast address formats.

Yes: ipintr recognizes the limited (all-1s and all-0s) and the network-directed and subnet-directed broadcast addresses.

- *Must* use an IP broadcast or IP multicast destination address in a link-layer broadcast.

Yes: ip\_output enables the link-layer multicast or broadcast flags only when the destination is an IP multicast or broadcast address.

- *Should* silently discard link-layer broadcasts when the packet does not specify an IP broadcast address as its destination.

No: There is no explicit test for the

`M_BCAST` or `M_MCAST` flags on incoming packets in Net/3, but `ip_forward` will discard these packets before forwarding them.

- *Should* use limited broadcast address for connected networks.

Partially: The decision to use the limited broadcast address (versus a subnet-directed or network-directed broadcast) is left to the application level by Net/3.

## IP Interface

- *Must* allow transport layer to use all IP mechanisms (e.g., IP options, TTL, TOS).

Yes: All the IP mechanisms are available to the transport layer in Net/3.

- *Must* pass interface identification up to transport layer.

Yes: The `m_pkthdr.rcvif` member of each mbuf containing an incoming packet points to the ifnet structure of

the interface that received the packet.

- *Must* pass all IP options to transport layer.

Yes: The entire IP header, including options, is present in the packet passed to the pr\_input function of the receiving transport protocol by ipintr.

- *Must* allow transport layer to send ICMP port unreachable and any of the ICMP query messages.

Yes: The transport layer may send any ICMP error messages by calling icmp\_error or may format and send any type of IP datagram by calling the ip\_output function.

- *Must* pass the following ICMP messages to the transport layer: destination unreachable, source quench, echo reply, timestamp reply, and time exceeded.

Yes: These messages are distributed by ICMP to other transport protocols or to any waiting processes using the raw IP

socket mechanism.

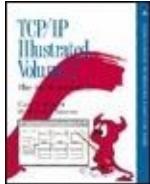
- *Must* include contents of ICMP message (IP header plus the data bytes present) in ICMP message passed to the transport layer.

Yes: icmp\_input passes the portion of the original IP packet contained within the ICMP message to the transport layers.

- *Should* be able to leap tall buildings at a single bound.

No: The next version of IP may meet this requirement.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.3 IP Options Requirements

This section summarizes the IP option processing requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* allow transport layer to send IP options.

Yes: The second argument to `ip_output` is a list of IP options to include in the outgoing IP datagram.

- *Must* pass all IP options received to higher layer.

Yes: The IP header and options are

passed to the pr\_input function of the receiving transport protocol.

- *Must* silently ignore unknown options.

Yes: The default case in ip\_dooptions skips over unknown options.

- *May* support the security option.

No: Net/3 does not support the IP security option.

- *Should not* send the stream identifier option and *must ignore* it in received datagrams.

Yes: Net/3 does not support the stream identifier option and ignores it on incoming datagrams.

- *May* support the record route option.

Yes: Net/3 supports the record route option.

- *May* support the timestamp option.

Partially: Net/3 supports the timestamp

option but does not implement it exactly as specified. The originating host does not insert a timestamp when required but the destination host records a timestamp before passing the datagram to the transport layer. The timestamp value follows the rules regarding standard values as specified in Section 3.2.2.8 of RFC 1122 for the ICMP timestamp message.

- *Must* support originating a source route and *must* be able to act as the final destination of a source route.

Yes: A source route may be included in the options passed to `ip_output`, and `ip_dooptions` correctly terminates a source route and saves it for use in constructing return routes.

- *Must* pass a datagram with completed source route up to the transport layer.

Yes: The source route option is passed up with any other options that may have appeared in the datagram.

- *Must* build correct (nonredundant) return route.

No: Net/3 blindly reverses the source route and does not check or correct for a route that was built incorrectly with a redundant hop for the original source host.

- *Must* not send multiple source route options in one header.

No: The IP layer in Net/3 does not prohibit a transport protocol from constructing and sending multiple source route options in a single datagram.

## Source Route Forwarding

- *May* support packet forwarding with the source route option.

Yes: Net/3 supports the source route options. `ip_dooptions` does all the work.

- *Must* obey corresponding router rules while processing source routes.

Yes: Net/3 follows the router rules whether or not the packet contains a source route.

- *Must* update TTL according to gateway rules.

Yes: ip\_forward implements this requirement.

- *Must* generate ICMP error codes 4 and 5 (fragmentation required and source route failed).

Yes: ip\_output is able to generate a fragmentation required message, and ip\_dooptions is able to generate the source route failed message.

- *Must* allow the IP source address of a source routed packet to not be an IP address of the forwarding host.

Yes: ip\_output transmits such packets.

RFC 1122 lists this as a *may* requirement because the addresses *may* be different, which *must* be allowed.

- *Must* update timestamp and record route options.

Yes: ip\_dooptions processes these options for source routed packets.

- *Must* support a configurable switch for *nonlocal source routing*. The switch *must* default to off.

No: Net/3 always allows nonlocal source routing and does not provide a switch to disable this function. Nonlocal source routing is routing packets between two different interfaces instead of receiving and sending the packet on the same interface.

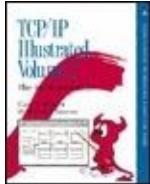
- *Must* satisfy gateway access rules for nonlocal source routing.

Yes: Net/3 follows the forwarding rules for nonlocal source routing.

- *Should* send an ICMP destination unreachable error (source route failed) if a source routed packet cannot be forwarded (except for ICMP error messages).

Yes: ip\_dooptions sends the ICMP destination unreachable error.  
icmp\_error discards it if the original datagram was an ICMP error message.

---



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.4 IP Fragmentation and Reassembly Requirements

This section summarizes the IP fragmentation and reassembly requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* be able to reassemble incoming datagrams of at least 576 bytes.

Yes: `ip_reass` supports reassembly of datagrams of indefinite size.

- *Should* support a configurable or indefinite maximum size for incoming

datagrams.

Yes: Net/3 supports an indefinite maximum size for incoming datagrams.

- *Must* provide a mechanism for the transport layer to learn the maximum datagram size to receive.

Not applicable: Net/3 has an indefinite limit based on available memory.

- *Must* send ICMP time exceeded error on reassembly timeout.

No: Net/3 does not send an ICMP time exceeded error. See [Figure 10.30](#) and [Exercise 10.1](#).

- *Should* support a fixed reassembly timeout value. The remaining TTL value in a received IP fragment *should not* be used as a reassembly timeout value.

Yes: Net/3 uses a compile-time value of 30 seconds (IPFRAGTTL is 60 slow-timeout intervals, which equals 30 seconds).

- *Must* provide the MMS\_S (maximum message size to send) to higher layers.

Partially: TCP derives the MMS\_S from the MTU found in the route entry for the destination or from the MTU of the outgoing interface. A UDP application does not have access to this information.

- *May* support local fragmentation of outgoing packets.

Yes: ip\_output fragments an outgoing packet if it is too large for the selected interface.

- *Must* not allow transport layer to send a message larger than MMS\_S if local fragmentation is not supported.

Not applicable: This is a transport-level requirement that does not apply to Net/3 since local fragmentation is supported.

- *Should not* send messages larger than 576 bytes to a remote destination in the absence of other information

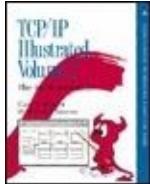
regarding the path MTU to the destination.

Partially: Net/3 TCP defaults to a segment size of 552 (512 data bytes + 40 header bytes). Net/3 UDP applications cannot determine if a destination is local or remote and so they often restrict their messages to 540 bytes (512 + 20 + 8). There is no kernel mechanism that prohibits sending larger messages.

- May support an all-subnets-MTU configuration flag.

Yes: The global integer subnetsarelocal defaults to true. TCP uses this flag to select a larger segment size (the size of the outgoing interface's MTU) instead of the default segment size for destinations on a subnet of the local network.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.5 ICMP Requirements

This section summarizes the ICMP requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* silently discard ICMP messages with unknown type.

Partially: icmp\_input ignores these messages and passes them to rip\_input, which delivers the message to any waiting processes or silently discards the message if no process is prepared to receive the message.

- *May* include more than 8 bytes of the

original datagram.

No: The icmp\_error function returns only a maximum of 8 bytes of the original datagram in the ICMP error message, Exercise 11.9.

- *Must* return the header and data unchanged from the received datagram.

Partially: Net/3 converts the ID, offset, and length fields of an IP packet from network byte order to host byte order in ipintr. This facilitates processing the packet, but Net/3 neglects to return the offset and length fields to network byte order before including the header in an ICMP error message. If the system operates with the same byte ordering as the network, this error is harmless. If it operates with a different ordering, the IP header contained within the ICMP error message has incorrect offset and length values.

The authors found that an Intel implementation of SVR4 and AIX 3.2 (Net/2 based) both return the length

byte-swapped. Implementations other than Net/2 or Net/3 that were tried (Cisco, NetBlazer, VM, and Solaris 2.3) did not have this bug.

Another error occurs when an ICMP port unreachable error is sent from the UDP code: the header length of the received datagram is changed incorrectly ([Section 23.7](#)). The authors found this error in Net/2 and Net/3 implementations. Net/1, however, did not have the bug.

- *Must* demultiplex received ICMP error message to transport protocol.  
Yes: icmp\_error uses the protocol field from the original header to select the appropriate transport protocol to respond to the error.
- *Should* send ICMP error messages with a TOS field of 0.  
Yes: All ICMP error messages are constructed with a TOS of 0 by icmp\_error.

- *Must not* send an ICMP error message caused by a previous ICMP error message.

Partially: icmp\_error sends an error for an ICMP redirect message, which Section 3.2.2 of RFC 1122 classifies as an ICMP error message.

- *Must not* send an ICMP error message caused by an IP broadcast or IP multicast datagram.

No: icmp\_error does not check for this case.

The icmp\_error function from the original Deering multicast code for BSD checks for this case.

- *Must not* send an ICMP error message caused by a link-layer broadcast.

Yes: icmp\_error discards ICMP messages in response to packets that arrived as link-layer broadcasts or multicasts.

- *Must not* send an ICMP error message

caused by a noninitial fragment.

Yes: icmp\_error discards errors generated in this case.

- *Must not* send an ICMP error message caused by a datagram with nonunique source address.

Yes: icmp\_reflect checks for experimental and multicast addresses. ip\_output discards messages sent from a broadcast address.

- *Must* return ICMP error messages when not prohibited.

Partially: In general, Net/3 sends appropriate ICMP error messages. It fails to send an ICMP reassembly timeout message at the appropriate time ([Exercise 10.1](#)).

- *Should* generate ICMP destination unreachable (protocol and port).

Partially: Datagrams for unsupported protocols are delivered to rip\_input where they are silently discarded if

there are no processes registered to accept the datagrams. UDP generates an ICMP port unreachable error.

- *Must* pass ICMP destination unreachable to higher layer.

Yes: icmp\_input passes the message to the pr\_ctlinput function defined for the protocol (udp\_ctlinput and tcp\_ctlinput for UDP and TCP, respectively).

- *Should* respond to destination unreachable error.

See [Sections 23.9 and 27.6](#).

- *Must* interpret destination unreachable as only a hint, as it may indicate a transient condition.

See [Sections 23.9 and 27.6](#).

- *Must not* send an ICMP redirect when configured as a host.

Yes: ip\_forward, the only function that detects and sends redirects, is not called unless the system is configured

as a router.

- *Must* update route cache when an ICMP redirect is received.

Yes: ipintr calls rtredirect to process the message.

- *Must* handle both host and network redirects. Furthermore, network redirects must be treated as host redirects.

Yes: ipintr calls rtredirect for both types of messages.

- *Should* discard illegal redirects.

Yes: rtredirect discards illegal redirects ([Section 19.7](#)).

- *May* send source quench if memory is unavailable.

Yes: ip\_forward sends a source quench if ip\_output returns ENOBUFS. This occurs when there is a shortage of mbufs or when an interface output queue is full.

- *Must* pass source quench to higher layer.

Yes: `icmp_input` passes source quench errors to the transport layers.

- *Should* respond to source quench in higher layer.

See [Sections 23.9 and 27.6](#) for UDP and TCP processing. Neither ICMP nor IGMP accept ICMP error messages (they don't define a `pr_ctlinput` function), in which case they are discarded by IP.

- *Must* pass time exceeded error to transport layer.

Yes: `icmp_input` passes this message to the transport layers.

- *Should* send parameter problem errors.

Yes: `ip_dooptions` complains about incorrectly formed options.

- *Must* pass parameter problem errors to transport layer.

Yes: icmp\_input passes parameter problem errors to the transport layer.

- *May* report parameter problem errors to process.

See [Sections 23.9](#) and [27.6](#) for UDP and TCP processing. Neither ICMP nor IGMP accept ICMP error messages.

- *Must* support an echo server and *should* support an echo client.

Yes: icmp\_input implements the echo server and the ping program implements the echo client using a raw IP socket.

- *May* discard echo requests to a broadcast address.

No: The reply is sent by icmp\_reflect.

- *May* discard echo request to multicast address.

No: Net/3 responds to multicast echo requests. Both icmp\_reflect and ip\_output permit multicast destination

addresses.

- *Must* use specific destination address as echo reply source.

Yes: icmp\_reflect converts a broadcast or multicast destination to the specific address of the receiving interface and uses the result as the source address for the echo reply.

- *Must* return echo request data in echo reply.

Yes: The data portion of the echo request is not altered by icmp\_reflect.

- *Must* pass echo reply to higher layer.

Yes: ICMP echo replies are passed to rip\_input for receipt by registered processes.

- *Must* reflect record route and timestamp options in ICMP echo request message.

Yes: icmp\_reflect includes the record route and timestamp options in the

echo reply message.

- *Must* reverse and reflect source route option.

Yes: icmp\_reflect retrieves the reversed source route with ip\_srcroute and includes it in the outgoing echo reply.

- *Should not* support the ICMP information request or reply.

Partially: The kernel does not generate or respond to either message, but a process may send or receive the messages through the raw IP mechanism.

- *May* implement the ICMP timestamp request and timestamp reply messages.

Yes: icmp\_input implements the timestamp server functionality. The timestamp client may be implemented through the raw IP mechanism.

- *Must* minimize timestamp delay variability (if implementing the timestamp messages).

Partially: The receive timestamp is applied after the message is taken off the IP input queue and the transmit timestamp is applied before the message is placed in the interface output queue.

- *May* silently discard broadcast timestamp request.

No: icmp\_input responds to broadcast timestamp requests.

- *May* silently discard multicast timestamp requests.

No: icmp\_input responds to broadcast timestamp requests.

- *Must* use specific destination address as timestamp reply source address.

Yes: icmp\_reflect converts a broadcast or multicast destination to the specific address of the receiving interface and uses the result as the source address for the timestamp reply.

- *Should* reflect record route and

timestamp options in an ICMP timestamp request.

Yes: icmp\_reflect includes the record route and timestamp options in the timestamp reply message.

- *Must* reverse and reflect source route option in ICMP timestamp request.

Yes: icmp\_reflect retrieves the reversed source route with ip\_srcroute and includes it in the outgoing timestamp reply.

- *Must* pass timestamp reply to higher layer.

Yes: ICMP timestamp replies are passed to rip\_input for receipt by registered processes.

- *Must* obey rules for standard timestamp value.

Yes: icmp\_input calls iptime, which returns a standard time value.

- *Must* provide a configurable method for

selecting the address mask selection method for an interface.

No: Net/3 supports only static configuration of address masks through the ifconfig program.

- *Must* support static configuration of address mask.

Yes: This is accomplished indirectly by specifying static information when the ifconfig program configures an interface during system initialization, typically in the /etc/netstart start-up script.

- *May* get address mask dynamically during system initialization.

No: Net/3 does not support the use of BOOTP or DHCP to acquire address mask information.

- *May* get address with an ICMP address mask request and reply messages.

No: Net/3 does not support the use ICMP messages to acquire address mask information.

- *Must* retransmit address mask request if no reply.

Not Applicable: Not required since this method is not implemented by Net/3.

- *Should* assume default mask if no reply is received.

Not Applicable: Not required since this method is not implemented by Net/3.

- *Must* update address mask from first reply only.

Not Applicable: Not required since this method is not implemented by Net/3.

- *Should* perform reasonableness check on any installed address mask.

No: Net/3 performs no reasonableness check on address masks.

- *Must not* send unauthorized address mask reply messages and *must* be explicitly configured to be agent.

Yes: icmp\_input only responds to

address mask requests if `icmpmaskrepl` is nonzero (it defaults to 0).

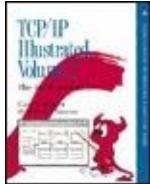
- *Should* support an associated address mask authority flag with each static address mask configuration.

No: Net/3 consults a global authority flag (`icmpmaskrepl`) to determine if it should send address mask replies for *any* interface.

- *Must* broadcast address mask reply when initialized.

No: Net/3 does not broadcast an address mask reply when an interface is configured.





[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.6 Multicasting Requirements

This section summarizes the IP multicast requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Should* support local IP multicasting (RFC 1112).

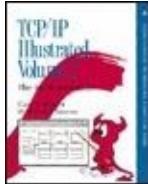
Yes: Net/3 supports IP multicasting.

- *Should* join the all-hosts group at start-up.

Yes: `in_ifinit` joins the all-hosts group while initializing an interface.

- *Should* provide a mechanism for higher layers to discover an interface's IP multicast capability.

Yes: The IFF\_MULTICAST flag in the interface's ifnet structure is available directly to kernel code and by the SIOCGIFFLAGS command for processes.



[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

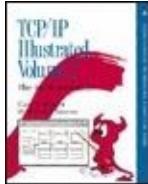
## Appendix C. RFC 1122 Compliance

### C.7 IGMP Requirements

This section summarizes the IGMP requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *May support IGMP (RFC 1112).*

Yes: Net/3 supports IGMP.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.8 Routing Requirements

This section summarizes the routing requirements from Section 3.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements. Be aware that the requirements of this RFC apply to a host and not necessarily the kernel implementation. Some items are not explicitly handled by the kernel routing function in Net/3, but they are expected to be provided by a routing daemon such as routed or gated.

- Must use address mask in determining whether a datagram's destination is on a connected network.

Yes: When an interface for a connected network such as an Ethernet is configured, its address mask is specified (or a default is chosen based on the class of IP address) and stored in the routing table entry. This mask is used by `rn_match` when it checks a leaf for a network match.

- *Must* operate correctly in a minimal environment when there are no routers (all networks are directly connected).

Yes: The system administrator must not configure a default route in this case.

- *Must* keep a "route cache" of mappings to next-hop routers.

Yes: The routing table is the cache.

- *Should* treat a received network redirect the same as a host redirect.

Yes, as described in [Section 19.7](#).

- *Must* use a default router when no entry exists for the destination in the routing table.

Yes, if a default route has been entered into the routing table.

- *Must* support multiple default routers.

Multiple defaults are not supported by the kernel. Instead, this should be provided by a routing daemon.

- *May* implement a table of static routes.

Yes: These can be created at system initialization time with the route command.

- *May* include a flag with each static route specifying whether or not the route can be overridden by a redirect.

No.

- *May* allow the routing table key to be a complete host address and not just a network address.

Yes: Host routes take priority over a network route to the same network.

- *Should* include the TOS in the routing

table entry.

No: There is a TOS field in the `sockaddr_inarp` that we describe in [Chapter 21](#), but it is not currently used.

- *Must* be able to detect the failure of a next-hop router that appears as the gateway field in the routing table and be able to choose an alternate next-hop router.

Negative advice, the `RTM_LOSING` message generated by `in_losing`, is passed to any processes reading from a routing socket, which allows the process (e.g., a routing daemon) to handle this event.

- *Should not* assume that a route is good forever.

Yes: There are no timeouts on routing table entries in the kernel other than those created by ARP Again, the standard Unix routing daemons time out routes and replace them with alternatives when possible.

- *Must not* ping routers continuously (ICMP echo request).

Yes: The Net/3 kernel does not do this.  
The routing daemons don't generate  
ICMP echo requests either.

- *Must* use pinging of a router only when traffic is being sent to that router.

The Net/3 kernel never generates pings to a next-hop router.

- *Should* allow higher and lower layers to give positive and negative advice.

Partially: The only information passed by other layers to the Net/3 routing functions is by `in_losing`, which is called only from TCP. The only action performed by the routing layer is to generate the `RTM_LOSING` message.

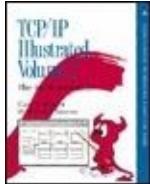
- *Must* switch to another default router when the existing default fails.

Yes, although the Net/3 kernel does not do this, it is supported by the routing daemons.

- *Must* allow the following information to be configured manually in the routing table: IP address, network mask, list of defaults.

Yes, but only one default is supported in the kernel.





[TCP/IP Illustrated, Volume 2: The Implementation](#) By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.9 ARP Requirements

This section summarizes the ARP requirements from Section 2.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Must* provide a mechanism to flush out-of-date ARP entries. If this mechanism involves a timeout, it *should* be configurable.

Yes and yes: arptimer provides this mechanism. The timeout is configurable (the arpt\_prune and arpt\_keep globals) but the only ways to change their values are to recompile the kernel or

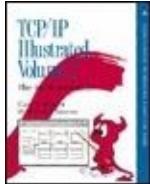
modify the kernel with a debugger.

- *Must* include a mechanism to prevent ARP flooding.

Yes, as we described with [Figure 21.24](#).

- *Should* save (rather than discard) at least one (the latest) packet of each set of packets destined to the same unresolved IP address, and transmit the saved packet when the address has been resolved.

Yes: This is the purpose of the `la_hold` member of the `llinfo_arp` structure.



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.10 UDP Requirements

This section summarizes the UDP requirements from Section 4.1.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

- *Should* send ICMP port unreachable.

Yes: `udp_input` does this.

- *Must* pass received IP options to application.

No: The code to do this is commented out in `udp_input`. This means that a process that receives a UDP datagram with a source route option cannot send

a reply using the reversed route.

- *Must* allow application to specify IP options to send.

Yes: The IP\_OPTIONS socket option does this. The options are saved in the PCB and placed into the outgoing IP datagram by ip\_output.

- *Must* pass IP options down to IP layer.

Yes: As mentioned above, IP places the options into the IP datagram.

- *Must* pass received ICMP messages to application.

Yes: We must look at the exact wording from the RFC: "A UDP-based application that wants to receive ICMP error messages is responsible for maintaining the state necessary to demultiplex these messages when they arrive; for example, the application may keep a pending receive operation for this purpose." The state required by Berkeley-derived systems is that the socket be connected to the foreign

address and port. As the comments at the beginning of [Figure 23.26](#) indicate, some applications create both a connected and an unconnected socket for a given foreign port, using the connected socket to receive asynchronous errors.

- *Must* be able to generate and verify UDP checksum.

Yes: This is done by `udp_input`, based on the global integer `udpcksum`.

- *Must* silently discard datagrams with bad checksum.

Yes: This is done only if `udpcksum` is nonzero. As we mentioned earlier, this variable controls both the sending of checksums and the verification of received checksums. If this variable is 0, the kernel does not verify a received nonzero checksum.

- *May* allow sending application to specify whether outgoing checksum is calculated, but *must* default to on.

No: The application has no control over UDP checksums. Regarding the default, UDP checksums are generated unless the kernel is compiled with 4.2BSD compatibility defined, or unless the administrator has disabled UDP checksums using sysctl(8).

- *May* allow receiving application to specify whether received UDP datagrams without a checksum (i.e., the received checksum is 0) are discarded or passed to the application.

No: Received datagrams with a checksum field of 0 are passed to the receiving process.

- *Must* pass destination IP address to application.

Yes: The application must call recvmsg and specify the IP\_RECVSTADDR socket option. Also recall our discussion following [Figure 23.25](#) noting that 4.4BSD broke this option when the destination address is a multicast or broadcast address.

- *Must* allow application to specify local IP address to be used when sending a UDP datagram.

Yes: The application can call bind to set the local IP address. Recall our discussion at the end of [Section 22.8](#) about the difference between the source IP address and the IP address of the outgoing interface. Net/3 does not allow the application to choose the outgoing interface that is done by ip\_output, based on the route to the destination IP address.

- *Must* allow application to specify wildcard local IP address.

Yes: If the IP address INADDR\_ANY is specified in the call to bind, the local IP address is chosen by in\_pcbconnect, based on the route to the destination.

- *Should* allow application to learn of the local address that was chosen.

Yes: The application must call connect. When a datagram is sent on an

unconnected socket with a wildcard local address, `ip_output` chooses the outgoing interface, which also becomes the source address. The `inp_laddr` member of the PCB, however, is restored to the wildcard address at the end of `udp_output` before `sendto` returns. Therefore, `getsockname` cannot return the value. But the application can connect a UDP socket to the destination, causing `in_pcbconnect` to determine the local interface and store the address in the PCB. The application can then call `getsockname` to fetch the IP address of the local interface.

- *Must* silently discard a received UDP datagram with an invalid source IP address (broadcast or multicast).

No: A received UDP datagram with an invalid source address is delivered to a socket, if a socket is bound to the destination port.

- *Must* send a valid IP source address.

Yes: If the local IP address is set by

bind, it checks the validity of the address. If the local IP address is wildcarded, ip\_output chooses the local address.

- *Must* provide the full IP interface from Section 3.4 of RFC 1122.

Refer to [Section C.2](#).

- *Must* allow application to specify TTL, TOS, and IP options for output datagrams.

Yes: The application can use the IP\_TTL, IP\_TOS, and IP\_OPTIONS socket options.

- *May* pass received TOS to application.

No: There is no way for the application to receive this value from the IP header. Notice that a getsockopt of IP\_TOS returns the value used in outgoing datagrams, not the value from a received datagram. The received ip\_tos value is available to udp\_input, but is discarded along with the entire IP header.

---

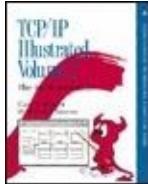
**Team-Fly**



[◀ Previous](#)

[Next ▶](#)

[Top](#)



[TCP/IP Illustrated, Volume 2: The Implementation](#)  
By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

## Appendix C. RFC 1122 Compliance

### C.11 TCP Requirements

This section summarizes the TCP requirements from Section 4.2.5 of RFC 1122 and the compliance of the Net/3 code that we've examined to those requirements.

#### PSH Flag

- *May aggregate data sent by the user without the PSH flag.*

Yes and no: Net/3 does not give the process a way to specify the PSH flag with a write operation, but Net/3 does aggregate data sent by the user in separate write operations.

- *May* queue data received without the PSH flag.

No: The absence or presence of a PSH flag in a received datagram makes no difference. Received data is placed onto the socket's received queue when it is processed.

- Sender *should* collapse successive PSH flags when it packetizes data.

No.

- *May* implement PSH flag on write calls.

No: This is not part of the sockets API.

- Since the PSH flag is not part of the write calls, *must not* buffer data indefinitely and *must* set the PSH flag in the last buffered segment.

Yes: This is the method used by Berkeley-derived implementations.

- *May* pass received PSH flag to application.

No: This is not part of the sockets API.

- *Should* send maximum-sized segment whenever possible, to improve performance.

Yes.

## Window

- *Must* treat window size as an unsigned number. *Should* treat window size as 32-bit value.

Yes: All the window sizes in [Figure 24.13](#) are unsigned longs, which is also required by the window scale option of RFC 1323.

- Receiver *must not* shrink the window (move the right edge to the left).

Yes, in [Figure 26.29](#).

- Sender *must* be robust against window shrinking.

Yes, in [Figure 29.15](#).

- *May* keep offered receive window closed indefinitely.

Yes.

- Sender *must* probe a zero window.

Yes, this is the purpose of the persist timer.

- *Should* send first zero-window probe when the window has been closed for the RTO.

No: Net/3 sets a lower bound for the persist timer of 5 seconds, which is normally greater than the RTO.

- *Should* exponentially increase the interval between successive probes.

Yes, as shown in [Figure 25.14](#).

- *Must* allow peer's window to stay closed indefinitely.

Yes, TCP never gives up probing a closed window.

- Sender *must not* timeout a connection just because the other end keeps advertising a zero window.

Yes.

## Urgent Data

- *Must* have urgent pointer point to last byte of urgent data.  
No: Berkeley-derived implementations continue to interpret the urgent pointer as pointing just beyond the last byte of urgent data.
- *Must* support a sequence of urgent data of any length.

Yes, with the bug fix discussed in Exercise 26.6.

- *Must* inform the receiving process (1) when TCP receives an urgent pointer and there was no previously pending urgent data, or (2) when the urgent pointer advances in the data stream.

Yes, in [Figure 29.17](#).

- *Must* be a way for the process to determine how much urgent data remains, or at least whether more urgent data remains to be read.

Yes, this is the purpose of the out-of-band mark, the SIOCATMARK ioctl.

## TCP Options

- *Must* be able to receive TCP options in any segment.

Yes.

- *Must* ignore any options not supported.

Yes, in [Section 28.3](#).

- *Must* cope with an illegal option length.

Yes, in [Section 28.3](#).

- *Must* implement both sending and receiving the MSS option.

Yes, a received MSS option is handled in [Figure 28.10](#), and [Figure 26.23](#) always sends an MSS option with a SYN.

- *Should* send an MSS option in every SYN when its receive MSS differs from 536, and *may* send it always.

Yes, as mentioned earlier, an MSS option is always sent by Net/3 with a SYN.

- If an MSS option is not received with a SYN, *must* assume a default MSS of 536.

No: The default MSS is 512, not 536.

This is probably a historical artifact because VAXes had a physical page size of 512 bytes and trailer protocols working only with data that is a multiple of 512.

- *Must* calculate the "effective send MSS."

Yes, in [Section 27.5](#).

## TCP Checksums

- *Must* generate a TCP checksum in outgoing segments and *must* verify received checksums.

Yes, TCP checksums are always calculated and verified.

## Initial Sequence Number Selection

- *Must* use the specified clock-driven selection from RFC 793.

No: RFC 793 specifies a clock that changes by 125,000 every half-second, whereas the Net/3 ISN (the global variable `tcp_iss`) is incremented by 64,000 every half-second, about one-half the specified rate.

## Opening Connections

- *Must* support simultaneous open attempts.

Yes, although Berkeley-derived systems

prior to 4.4BSD did not support this, as described in [Section 28.9](#).

- *Must* keep track of whether it reached the SYN\_RCVD state from the LISTEN or SYN\_SENT states.

Yes, same result, different technique. The purpose of this requirement is to allow a passive open that receives an RST to return to the LISTEN state (as shown in [Figure 24.15](#)), but force an active open that ends up in SYN\_RCVD and then receives an RST to be aborted. This is described following [Figure 28.36](#).

- A passive open *must not* affect previously created connections.

Yes.

- *Must* allow a listening socket with a given local port at the same time that another socket with the same local port is in the SYN\_SENT or SYN\_RCVD state.

Yes: The stated purpose of this requirement is to allow a given

application to accept multiple connection attempts at about the same time. This is done in Berkeley-derived implementations by cloning new connections from the socket in the LISTEN state when the incoming SYN arrives.

- *Must* ask IP to select a local IP address to be used as the source IP address when the source IP address is not specified by the process performing an active open on a multihomed host.

Yes, done by `in_pcbsconnect`.

- *Must* continue to use the same source IP address for all segments sent on a connection.

Yes: Once `in_pcbsconnect` selects the source address, it doesn't change.

- *Must not* allow an active open for a broadcast or multicast foreign address.

Yes and no: TCP will not send segments to a broadcast address because the call to `ip_output` in [Figure 26.32](#) does not

specify the SO\_BROADCAST option. Net/3, however, allows connection attempts to multicast addresses.

- *Must* ignore incoming SYNs with an invalid source address.

Yes: The code in [Figure 28.16](#) checks for these invalid source addresses.

## Closing Connections

- *Should* allow an RST to contain data.

No: The RST processing in [Figure 28.36](#) ends up jumping to drop, which skips the processing of any segment data in [Figure 29.22](#).

- *Must* inform process whether other end closed the connection normally (e.g., sent a FIN) or aborted the connection with an RST.

Yes: The read system calls return 0 (end-of-file) when the FIN is processed, but 1 with an error of ECONNRESET when an RST is received.

- *May* implement a half-close.

Yes: The process calls shutdown with a second argument of 1 to send a FIN. The process can still read from the connection.

- If the process completely closes a connection (i.e., not a half-close) and received data is still pending in TCP, or if new data arrives after the close, TCP *should* send an RST to indicate data was lost.

No and yes: If a process calls close and unread data is in the socket's receive buffer, an RST is not sent. But if data arrives after a socket is closed, an RST is returned to the sender.

- *Must* linger in TIME\_WAIT state for twice the MSL.

Yes, although the Net/3 MSL of 30 seconds is much smaller than the RFC 793 recommended value of 2 minutes.

- *May* accept a new SYN from a peer to reopen a connection directly from the

TIME\_WAIT state.

Yes, as shown in [Figure 28.29](#).

## Retransmissions

- *Must* implement Van Jacobson's slow start and congestion avoidance.

Yes.

- *May* reuse the same IP identifier field when a retransmission is identical to the original packet.

No: The IP identifier is assigned by `ip_output` from the global variable `ip_id`, which increments each time an IP datagram is sent. It is not assigned by TCP.

- *Must* implement Jacobson's algorithm for calculating the RTO and Karn's algorithm for selecting the RTT measurements.

Yes, but realize that when RFC 1323 timestamps are present, the

retransmission ambiguity problem is gone, obviating half of Karn's algorithm, as we discussed with [Figure 29.6](#).

- *Must* include an exponential backoff for successive RTO values.

Yes, as described with [Figure 25.22](#).

- Retransmission of SYN segments *should* use the same algorithm as data segments.

Yes, as shown in [Figure 25.15](#).

- *Should* initialize estimation parameters to calculate an initial RTO of 3 seconds.

No: The initial value of `t_rxtcur` calculated by `tcp_newtcpcb` is 6 seconds. This is also seen in [Figure 25.15](#).

- *Should* have a lower bound on the RTO measured in fractions of a second and an upper bound of twice the MSL.

No: The lower bound is 1 second and the upper bound is 64 seconds ([Figure](#)

25.3).

## Generating ACKs

- *Should* queue out-of-order segments.

Yes, done by `tcp_reass`.

- *Must* process all queued segments before sending any ACKs.

Yes, but only for in-order segments.

`ipintr` calls `tcp_input` for each queued datagram that is a TCP segment. For in-order segments, `tcp_input` schedules a delayed ACK and returns to `ipintr`. If there are additional TCP segments on IP's input queue, `tcp_input` is called by `ipintr` for each one. Only when `ipintr` finds no more IP datagrams on its input queue and returns can `tcp_fasttimo` be called to generate a delayed ACK. This ACK will contain the highest acknowledgment number in all the segments processed by `tcp_input`.

The problem is with out-of-order segments: `tcp_input` calls `tcp_output`

itself, before returning to ipintr, to generate the ACK for the out-of-order segment. If there are additional segments on IP's input queue that would have made the out-of-order segment be in order, they are processed after the immediate ACK is sent.

- *May* generate an immediate ACK for an out-of-order segment.

Yes, this is needed for the fast retransmit and fast recovery algorithms ([Section 29.4](#)).

- *Should* implement delayed ACKs and the delay *must* be less than 0.5 seconds.

Yes: The TF\_DELACK flag is checked by the tcp\_fasttimmo function every 200 ms.

- *Should* send an ACK for at least every second segment.

Yes, the code in [Figure 26.9](#) generates an ACK for every second segment. We also discussed that this happens only if the process receiving the data reads the

data as it arrives, since the calls to `tcp_output` that cause every other segment to be acknowledged are driven by the `PRU_RCVD` request.

- *Must* include silly window syndrome avoidance in the receiver.

Yes, as seen in [Figure 26.29](#).

## Sending Data

- The TTL value for TCP segments *must* be configurable.

Yes: The TTL is initialized to 64 (`IPDEFTTL`) by `tcp_newtcpcb`, but can then be changed by a process using the `IP_TTL` socket option.

- *Must* include sender silly window syndrome avoidance.

Yes, in [Figure 26.8](#).

- *Should* implement the Nagle algorithm.

Yes, in [Figure 26.8](#).

- Must allow a process to disable the Nagle algorithm on a given connection.

Yes, with the TCP\_NODELAY socket option.

## Connection Failures

- Must pass negative advice to IP when the number of retransmissions for a given segment exceeds some value R1.

Yes: The value of R1 is 4, and in [Figure 25.26](#), when the number of retransmissions exceeds 4, in\_losing is called.

- Must close a connection when the number of retransmissions for a given segment exceeds some value R2.

Yes: The value of R2 is 12 ([Figure 25.26](#)).

- Must allow process to set the value of R2.

No: The value 12 is hardcoded in [Figure](#)

## 25.26.

- *Should* inform the process when R1 is reached and before R2 is reached.

No.

- *Should* default R1 to at least 3 retransmissions and R2 to at least 100 seconds.

Yes: R1 is 4 retransmissions, and with a minimum RTO of 1 second, the `tcp_backoff` array ([Section 25.9](#)) guarantees a minimum value of R2 of over 500 seconds.

- Must handle SYN retransmissions in the same general way as data retransmissions.

Yes, but R1 is normally not reached for the retransmission of a SYN ([Figure 25.15](#)).

- *Must* set R2 to at least 3 minutes for a SYN.

No: R2 for a SYN is limited to 75

seconds by the connection-establishment timer ([Figure 25.15](#)).

## Keepalive Packets

- *May* provide keepalives.

Yes, they are provided.

- *Must* allow process to turn keepalives on or off, and *must* default to off.

Yes: Default is off and process must turn them on with the SO\_KEEPALIVE socket option.

- *Must* send keepalives only when connection is idle for a given period.

Yes.

- *Must* allow the keepalive interval to be configurable and *must* default to no less than 2 hours.

No and yes: The idle time before sending keepalive probes is not easily configurable, but it defaults to 2 hours.

If the default idle time is changed (by changing the global variable `tcp_keepidle`), it affects all users of the `keepalive` option on the host. It cannot be configured on a per-connection basis as many users would like.

- *Must not* interpret the failure to respond to any given probe as a dead connection.

Yes: Nine probes are sent before the connection is considered dead.

## IP Options

- *Must ignore* received IP options it doesn't understand.

Yes: This is done by the IP layer.

- *May support* the timestamp and record route options in received segments.

No: Net/3 only reflects these options for ICMP packets that are reflected back to the sender (`icmp_reflect`). `tcp_input` discards any received IP options by

calling ip\_strioptions in [Figure 28.2](#).

- *Must* allow process to specify a source route when a connection is actively opened, and this route must take precedence over a source route received for this connection.

Yes: The source route is specified with the IP\_OPTIONS socket option.

tcp\_input never looks at a received source route when the connection is actively opened.

- *Must* save a received source route in a connection that is passively opened and use the return route for all segments sent on this connection. If a different source route arrives in a later segment, the later route *should* override the earlier one.

Yes and no: [Figure 28.7](#) calls ip\_srcroute, but only when the SYN arrives for a listening socket. If a different source route arrives later, it is not used.

## Receiving ICMP Messages from IP

- Receipt of an ICMP source quench *should* trigger slow start.

Yes: The function `tcp_quench` is called by `tcp_ctlinput`.

- Receipt of a network unreachable, host unreachable, or source route failed *must not* cause TCP to abort the connection and the process *should* be informed.

Yes and no: As described following [Figure 27.12](#), Net/3 now completely ignores host unreachable and network unreachable errors for an established connection.

- Receipt of a protocol unreachable, port unreachable, or fragmentation required and DF bit set *should* abort an existing connection.

No: `tcp_notify` records these ICMP errors in `t_softerror`, which is reported to the process if the connection is

eventually dropped.

- *Should* handle time exceeded and parameter problem errors the same as required previously for network and host unreachable.

Yes: ICMP parameter problem errors are just recorded in `t_softerror` by `tcp_notify`. ICMP time exceeded errors are ignored by `tcp_ctlinput`. Neither type of ICMP error causes the connection to be aborted.

## Application Programming Interface

- *Must* be a method for reporting soft errors to the process, normally in an asynchronous fashion.

No: Soft errors are returned to the process if the connection is aborted.

- *Must* allow process to specify TOS for segments sent on a connection. *Should* let application change this during a connection's lifetime.

Yes to both, with the IP\_TOS socket option.

- *May* pass most recently received TOS to process.

No: There is no way to do this with the sockets API. Calling getsockopt for IP\_TOS returns only the current value being sent; it does not return the most recently received value.

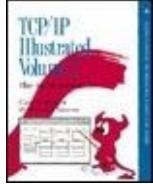
- *May* implement a "flush" call.

No: TCP sends the data from the process as quickly as it can.

- *Must* allow process to specify local IP address before either an active open or a passive open.

Yes: This is done by calling bind before either connect or accept.





**TCP/IP Illustrated, Volume 2: The Implementation** By Gary R. Wright,  
W. Richard Stevens  
[Table of Contents](#)

# Bibliography

All the RFCs are available at no charge through electronic mail or by using anonymous FTP across Internet as described in [Appendix B](#).

Whenever the authors were able to locate an electronic copy of papers and reports reference this bibliography, its URL (Uniform Resource Locator) ([Appendix B](#)) is included.

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC 1349, 28 pages (July).

Almquist, P., and Kastenholz, F. J. 1994. "Toward Requirements for IP Routers," RFC 1716, 186 pages (Nov.).

This RFC is an intermediate step to replace RFC [[Braden and Postel 1987](#)].

Auerbach, K. 1994. "Max IP Packet Length and Message-ID <karl.3.000A4DD7 @cavebear.com> Usenet, comp.protocols.tcp-ip Newsgroup (July)

Boggs, D. R. 1982. "Internet Broadcasting," Xerox PARC CSL-83-3, Stanford University, Palo Alto, (Jan.).

Braden, R. T., ed. 1989a. "Requirements for Internet Host Communication Layers," RFC 1122, 116 p (Oct.).

The first half of the Host Requirements RFC. This covers the link layer, IP, TCP, and UDP.

Braden, R. T., ed. 1989b. "Requirements for Internet Host Application and Support," RFC 1123, 98 p (Oct.).

The second half of the Host Requirements RFC. This half covers Telnet, FTP, TFTP, SMTP, and the DNS.

Braden, R. T. 1989c. "Perspective on the Host Requirements RFCs," RFC 1127, 20 pages (Oct.). An informal summary of the discussions and conclusions of the IETF working group that developed the Host Requirements RFC.

Braden, R. T. 1992. "TIME-WAIT Assassination Hazards in TCP," RFC 1337, 11 pages (May). Shows how the receipt of an RST while in the TIME\_WAIT state can lead to problems.

Braden, R. T. 1993. "TCP Extensions for High Performance: An Update," Internet Draft, 10 pages (June).

This is an update to RFC 1323 [[Jacobson, Braden, Borman 1992](#)].

<http://www.noao.edu/~rstevens/tcplw-extensi>

Braden, R. T. 1994. "T/TCP TCP Extensions for Transactions, Functional Specification," RFC 1644, 16 pages (July).

Braden, R. T., Borman, D. A., and Partridge, C. "Computing the Internet Checksum," RFC 1071, 16 pages (Sept.).

Provides techniques and algorithms for calculating the checksum used by IP, ICMP, IGMP, UDP, and TCP.

Braden, R.T., and Postel, J. B. 1987. "Requirements for Internet Gateways," RFC 1009, 55 pages (June). The equivalent of the Host Requirements RFC for routers. This RFC is being replaced by RFC 171 [Almquist and Kastenholz 1994].

Brakmo, L. S., O'Malley, S. W., and Peterson, L. 1994. "TCP Vegas: New Techniques for Congestions Detection and Avoidance," *Computer Communications Review*, vol. 24, no. 4, pp. 2435 (Oct.).

Describes modifications to the 4.3BSD Reno TCP implementation to improve throughput and reduce retransmissions.

<ftp://ftp.cs.arizona.edu/xkernel/Papers/vegas.pdf>

Carlson, J. 1993. "Re: Bug in Many Versions of

Message-ID  
<1993Jul12.130854.26176@xylogics.com>, Us  
comp.protocols.tcp-ip Newsgroup (July).

Casner, S., *Frequently Asked Questions (FAQ) c  
Multicast Backbone (MBONE)*, 1993.  
<ftp://ftp.isi.edu/mbone/faq.txt>

Cheswick, W. R., and Bellovin, S. M. 1994. *Fire  
and Internet Security: Repelling the Wily Hacke*  
Addison-Wesley, Reading, Mass.  
Describes how to set up and administer a firew  
gateway and the security issues involved.

Clark, D. D. 1982. "Modularity and Efficiency in  
Protocol Implementation," RFC 817, 26 pages (

Comer, D. E., and Lin, J. C. 1994. "TCP Bufferin  
Performance Over an ATM Network," Purdue Te  
Report CSD-TR 94-026, Purdue University, Wes  
Lafayette, Ind. (Mar.).

<ftp://gwen.cs.purdue.edu/pub/lin/TCP.atm.ps.Z>

Comer, D. E., and Stevens, D. L. 1993.  
*Internetworking with TCP/IP: Vol. III: ClientSei  
Programming and Applications, BSD Socket Ve*  
Prentice-Hall, Englewood Cliffs, N.J.

Croft, W., and Gilmore, J. 1985. "Bootstrap Prot

(BOOTP)," RFC 951, 12 pages (Sept.).

Crowcroft, J., Wakeman, I., Wang, Z., and Sirovich, L. 1992. "Is Layering Harmful?," *IEEE Network*, vol. 6, no. 1, pp. 20-24 (Jan.).

The seven missing figures from this paper appear in the next issue, vol. 6, no. 2 (March).

Dalton, C., Watson, G., Banks, D., Calamvokis, I., Edwards, A., and Lumley, J. 1993. "Afterburner," *IEEE Network*, vol. 7, no. 4, pp. 36-43 (July).

Describes how to speed up TCP by reducing the number of data copies performed, and a special purpose interface card that supports this design.

Deering, S. E. 1989. "Host Extensions for IP Multicasting," RFC 1112, 17 pages (Aug.).

The specification of IP multicasting and IGMP.

Deering, S. E., ed. 1991a. "ICMP Router Discovery and Information Exchange Messages," RFC 1256, 19 pages (Sept.).

Deering, S. E. 1991b. "Multicast Routing in a Datagram Internetwork," STAN-CS-92-1415, Stanford University, Palo Alto, Calif. (Dec.).

<ftp://gregorio.stanford.edu/vmtp-ip/sdthesis.part1.ps.Z>

Deering, S. E., and Cheriton, D. P. 1990. "Multicast Routing in a Datagram Internetwork,"

Routing in Datagram Internetworks and Extend LANs," *ACM Transactions on Computer Systems*: 8, no. 2, pp. 85110 (May).

Proposes extensions to common routing techniques to support multicasting.

Deering, S., Estrin, D., Farinacci, D., Jacobson, C., and Wei, L. 1994. "An Architecture for Wide Multicast Routing," *Computer Communication Review*: vol. 24, no. 4, pp. 126135 (Oct.).

Droms, R. 1993. "Dynamic Host Configuration Protocol," RFC 1541, 39 pages (Oct.).

Finlayson, R., Mann, T., Mogul, J. C., and Theimer, M. 1984. "A Reverse Address Resolution Protocol," IEN 903, 4 pages (June).

Floyd, S. 1994. Private Communication.

Forgie, J. 1979. "STA Proposed Internet Stream Protocol," IEN 119, MIT Lincoln Laboratory (Sep.).

Fuller, V., Li, T., Yu, J. Y., and Varadhan, K. 1992. "Classless Inter-Domain Routing (CIDR): An Architecture and Aggregation Strategy," RFC 1519, 15 pages (Sept.).

Hornig, C. 1984. "Standard for the Transmission of IP Datagrams over IEEE 802 Networks," RFC 913, 10 pages (Oct.).

Datagrams over Ethernet Networks," RFC 894, pages (Apr.).

Hutchinson, N. C., and Peterson, L. L. 1991. "The X Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 6476 (Jan.).  
<ftp://ftp.cs.arizona.edu/xkernel/Papers/architectural.pdf>

Itano, W. M., and Ramsey, N. F. 1993. "Accurate Measurement of Time," *Scientific American*, vol. 268, p. 56 (July).

Overview of historical and current methods for accurate timekeeping. Includes a short discussion of international time scales including International Atomic Time (TAI) and Coordinated Universal Time (UTC).

Jacobson, V. 1988a. "Some Interim Notes on the Network Speedup," Message-ID <8807200426.AA01221@helios.ee.lbl.gov>, Usenet comp.protocols.tcp-ip Newsgroup (July).

Jacobson, V. 1988b. "Congestion Avoidance and Control," *Computer Communication Review*, vol. 18, no. 4, pp. 314329 (Aug.).

A classic paper describing the slow start and congestion avoidance algorithms for TCP.

<ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z>

Jacobson, V. 1990a. "Compressing TCP/IP Headers for Low-Speed Serial Links," RFC 1144, 43 pages (Describes CSLIP, a version of SLIP with the TCP headers compressed.

Jacobson, V. 1990b. "4BSD TCP Header Prediction," *Computer Communication Review*, vol. 20, no. 1315 (Apr.).

Jacobson, V. 1990c. "Modified TCP Congestion Avoidance Algorithm," April 30, 1990, end2end mailing list (Apr.).

Describes the fast retransmit and fast recovery algorithms.

<ftp://ftp.isi.edu/end2end/end2end-interest-1990>

Jacobson, V. 1990d. "Berkeley TCP Evolution from Tahoe to 4.3-Reno," *Proceedings of the Eighteen Annual Internet Engineering Task Force*, p. 365 (Sept.) University of British Columbia, Vancouver, B.C.

Jacobson, V. 1993. "Some Design Issues for High-Speed Networks," *Networkshop '93* (Nov.), Melbourne, Australia.

A set of 21 overheads.

<ftp://ftp.ee.lbl.gov/talks/vj-nws93-1.ps.Z>

Jacobson, V., and Braden, R. T. 1988. "TCP Extensions for Long-Delay Paths," RFC 1072, 16 pages (October). Describes the selective acknowledgment option for TCP, which was removed from the later RFC 1323, along with the echo options, which were replaced with the timestamp option in RFC 1323.

Jacobson, V., Braden, R. T., and Borman, D. A. 1991. "TCP Extensions for High Performance," RFC 1326, 16 pages (May). Describes the window scale option, the timestamp option, and the PAWS algorithm, along with the reasons these modifications are needed. [Braden 1993] updates this RFC.

Jain, R., and Routhier, S. A. 1986. "Packet Train Measurements and a New Model for Computer Network Traffic," *IEEE Journal on Selected Areas in Communications*, vol. 4, pp. 1162-1167.

Karels, M. J., and McKusick, M. K. 1986. "Network Performance and Management with 4.3BSD and IP/TCP," *Proceedings of the 1986 Summer USENIX Conference*, pp. 182-188, Atlanta, Ga. Describes the changes made from 4.2BSD to 4.3BSD with regard to TCP/IP.

Karn, P., and Partridge, C. 1987. "Improving Router Performance," *IEEE Journal on Selected Areas in Communications*, vol. 5, pp. 1030-1039.

Trip Time Estimates in Reliable Transport Protocols," *Computer Communication Review*, vol. 17, no. 27 (Aug.).

Details of Karn's algorithm to handle the retransmission timeout for segments that have retransmitted.

<ftp://sics.se/users/craig/karn-partridge.ps>

Kay, J., and Pasquale, J. 1993. "The Importance of Non-Data Touching Processing Overheads in TCP," *Computer Communication Review*, vol. 23, no. 259268 (Sept.).

Kent, C. A., and Mogul, J. C. 1987. "Fragmentation Considered Harmful," *Computer Communication Review*, vol. 17, no. 5, pp. 390401 (Aug.).

Kernighan, B. W., and Plauger, P. J. 1976. *Software Tools*. Addison-Wesley, Reading, Mass.

Krol, E. 1994. *The Whole Internet, Second Edition*. O'Reilly & Associates, Sebastopol, Calif.  
An introduction into the Internet, common Internet applications, and various resources available on the Internet.

Krol, E., and Hoffman, E. 1993. "FYI on 'What is the Internet?'," RFC 1462, 11 pages (May).

Lanciani, D. 1993. "Re: Bug in Many Versions of the Internet Protocol Version 4," Message-ID <1993Jul10.015938.15951@burrhus.harvard.edu>, Usenet, comp.protocols.ip Newsgroup (July).

Leffler, S. J., McKusick, M. K., Karels, M.J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Mass.  
An entire book on the 4.3BSD Unix system. This describes the Tahoe release of 4.3BSD.

Lynch, D. C. 1993. "Historical Perspective," in *Internet System Handbook*, eds. D. C. Lynch and M. T. F. Eich, pp. 314. Addison-Wesley, Reading, Mass.  
A historical overview of the Internet and its predecessor, the ARPANET.

Mallory, T., and Kullberg, A. 1990. "Incremental Updating of the Internet Checksum," RFC 1141, 11 pages (Jan.).  
This RFC is updated by RFC 1624 [[Rijsinghani 1994](#)]

Mano, M. M. 1993. *Computer System Architecture: Principles and Design*, Third Edition. Prentice-Hall, Englewood Cliffs, NJ.

McCanne, S., and Jacobson, V. 1993. "The BSD Filter: A New Architecture for User-Level Packet Filtering,"

Capture," *Proceedings of the 1993 Winter USEI Conference*, pp. 259269, San Diego, Calif.

A detailed description of the BSD Packet Filter (and comparisons with Sun's Network Interface (NIT)).

<ftp://ftp.ee.lbl.gov/papers/bpf-usenix93.ps.Z>

McCloghrie, K., and Farinacci, D. 1994a. "Internet Group Management Protocol MIB," Internet Draft, 15 pages (Jul.).

McCloghrie, K., and Farinacci, D. 1994b. "IP Multicast Routing MIB," Internet Draft, 15 pages (Jul.).

McCloghrie, K., and Rose, M. T. 1991. "Management Information Base for Network Management of TCP-based Internets: MIB-II," RFC 1213 (Mar.).

McGregor, G. 1992. "PPP Internet Protocol Control Protocol (IPCP)," RFC 1332, 12 pages (May).

McKenney, P. E., and Dove, K. F. 1992. "Efficient Demultiplexing of Incoming TCP Packets," *Computer Communication Review*, vol. 22, no. 4, pp. 269 (Oct.).

Mogul, J. C. 1991. "Network Locality at the Source Processes," *Computer Communication Review*, no. 4, pp. 273284 (Sept.).

Mogul, J.C. 1993. "IP Network Performance," in *Internet System Handbook*, eds. D. C. Lynch and Rose, pp. 575675. Addison-Wesley, Reading, MA. Covers numerous topics in the Internet protocols that are candidates for tuning to obtain optimal performance.

Mogul, J. C., and Deering, S. E. 1990. "Path MTU Discovery," RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. "Internet Standard Subnetting Procedure," RFC 950, 18 pages (Aug.).

Moy, J. 1994. "Multicast Extensions to OSPF," RFC 1584, 102 pages (Mar.).

Olivier, G. 1994. "What is the Diameter of the Internet?," Message-ID  
<1994Jan22.094832@mines.u-nancy.fr>, User comp.unix.wizards Newsgroup (Jan.).

Partridge, C. 1987. "Implementing the Reliable Protocol (RDP)," *Proceedings of the 1987 Summer USENIX Conference*, pp. 367379, Phoenix, Arizona.

Partridge, C. 1993. "Jacobson on TCP in 30 Instructions," Message-ID  
<1993Sep8.213239.28992@sics.se>, Usenet,

comp.protocols.tcp-ip Newsgroup (Sept.). Describes a research implementation of TCP/IP developed by Van Jacobson that reduces TCP repacket processing down to 30 instructions on a system.

<http://www.kohala.com/~rstevens/vanj.93sep>

Partridge, C., and Hinden, R. 1990. "Version 2 of Reliable Data Protocol (RDP)," RFC 1151, 4 pages (Apr.).

Partridge, C., Mendez, T., and Milliken, W. 1993. "Anycasting Service," RFC 1546, 9 pages (Nov.)

Partridge, C., and Pink, S. 1993. "A Faster UDP," *IEEE/ACM Transactions on Networking*, vol. 1, issue 4, pp. 429-440 (Aug.).

Describes implementation improvements to the Berkeley sources to speed up UDP performance by 30%.

Paxson, V. 1994. Private Communication.

Perlman, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Mass.

Piscitello, D. M., and Chapin, A. L. 1993. *Open Systems Networking: TCP/IP and OSI*. Addison-Wesley, Reading, Mass.

Plummer, D. C. 1982. "An Ethernet Address Resolution Protocol," RFC 826, 10 pages (Nov.).

Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792, 21 pages (Sept.).

Postel, J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.).

Postel, J. B. 1981d. "Service Mappings," RFC 794, 15 pages (Sept.).

Postel, J. B., and Reynolds, J. K. 1988. "Standard for the Transmission of IP Datagrams over IEEE 802 Networks," RFC 1042, 15 pages (Apr.).

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, Mass.

Reynolds, J. K., and Postel, J. B. 1994. "Assigned Numbers," RFC 1700, 230 pages (Oct.).

Rijsinghani, A. 1994. "Computation of the Internet Checksum via Incremental Update," RFC 1624, 15 pages (May).

An update to RFC 1141 [[Mallory and Kullberg 1](#)

Romkey, J. L. 1988. "A Nonstandard for Transm  
of IP Datagrams Over Serial Lines: SLIP," RFC :  
pages (June).

Rose, M. T. 1990. *The Open Book: A Practical  
Perspective on OSI*. Prentice-Hall, Englewood C  
N.J.

Salus, P. H. 1994. *A Quarter Century of Unix*. A  
Wesley, Reading, Mass.

Sedgewick, R. 1990. *Algorithms in C*. Addison-W  
Reading, Mass.

Simpson, W.A. 1993. "The Point-to-Point Protoc  
(PPP)," RFC 1548, 53 pages (Dec.).

Sklower, K. 1991. "A Tree-Based Packet Routing  
for Berkeley Unix," *Proceedings of the 1991 Wi  
USENIX Conference*, pp. 9399, Dallas, Tex.

Stallings, W. 1987. *Handbook of Computer-  
Communications Standards, Volume 2: Local N  
Standards*. Macmillan, New York.

Stallings, W. 1993. *Networking Standards: A G  
OSI, ISDN, LAN, and MAN Standards*. Addison-  
Reading, Mass.

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, N.J.

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA.

Stevens, W. R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Mass. The first volume in this series, which provides a complete introduction to the Internet protocols.

Tanenbaum, A. S. 1989. *Computer Networks, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

Topolcic, C. 1990. "Experimental Stream Protocol Version 2 (SY-II)," RFC 1190, 148 pages (Oct.)

Torek, C. 1992. "Re: A Problem in Bind System Message-ID <27240@dog.ee.lbl.gov>, Usenet, comp.unix.internals Newsgroup (Nov.)."

Waitzman, D., Partridge, C., Deering, S. E. 1988. "Distance Vector Multicast Routing Protocol," RFC 1075, 24 pages (Nov.).